Doctoral Thesis

Evaluation and Optimization of Big Data Processing on High Performance Computing Systems

Jorge Veiga Fachal

2018

UNIVERSIDADE DA CORUÑA

Evaluation and Optimization of Big Data Processing on High Performance Computing Systems

Jorge Veiga Fachal

Doctoral Thesis

October 2018

PhD Advisors:

Roberto Rey Expósito
Juan Touriño Domínguez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dr. Roberto Rey Expósito
Profesor Ayudante Doctor
Dpto. de Ingeniería de Computadores
Universidade da Coruña

Dr. Juan Touriño Domínguez
Catedrático de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Evaluation and Optimization of Big Data Processing on High Performance Computing Systems*" ha sido realizada por D. Jorge Veiga Fachal bajo nuestra dirección en el Departamento de Ingeniería de Computadores de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 5 de Octubre de 2018

Fdo.: Roberto Rey Expósito
Director de la Tesis Doctoral

Fdo.: Juan Touriño Domínguez
Director de la Tesis Doctoral

Fdo.: Jorge Veiga Fachal
Autor de la Tesis Doctoral

A todos los que me habéis ayudado a realizar esta tesis,
y a los que no, pues mira, también

Acknowledgments

I want to thank my advisors Juan and Roberto for giving me the opportunity to develop this Thesis and for their valuable guidance; I believe working with them has been a truly enriching experience. I also thank all the members of the GAC that contributed to this work in some way, especially Xoan for helping me to take my first steps into the research world. Moreover, thanks to my lab colleagues for the wonderful work environment and for all the coffee breaks and dinners that we shared.

I want to say thanks to my family; to my parents for encouraging me to aim high, to my siblings Mar and Carlos for their support and to my beloved partner Ian for his company through all these years. I am very grateful to all my friends that lent me a hand when I needed it, especially Nuria, Manu and Laura for all the times they made me forget the work for a while.

I would like to thank Dr. Bruno Raffin and his group for hosting me during my three-month research visit to the Inria Grenoble Rhône-Alpes reseach center, and for providing access to the Grid'5000 computing platform. I really appreciated the comments on my work and the welcoming environment that I found there.

Finally, I want to acknowledge the following funders of this work: the Computer Architecture Group, the Department of Computer Engineering, and the University of A Coruña for the human and material support; the NESUS network under the COST Action IC1305; the Galician Government (refs. GRC2013/055, ED431C 2017/04, R2014/041, ED431D R2016/045 and predoctoral grant ED 481A-2015/189); and the Spanish Government (refs. TIN2013-42148-P, TIN2016-75845-P, FPU grant FPU14/02805 and mobility grant EST16/00076).

Jorge Veiga Fachal

A person has the right,
and I think the responsibility,
to develop all of their talents.

Jessye Norman

Resumo

Hoxe en día, moitas organizacións empregan tecnoloxías Big Data para extraer información de grandes volumes de datos. A medida que o tamaño destes volumes crece, satisfacer as demandas de rendemento das aplicacións de procesamento de datos masivos faise máis difícil. Esta Tese céntrase en avaliar e optimizar estas aplicacións, presentando dúas novas ferramentas chamadas BDEv e Flame-MR. Por unha banda, BDEv analiza o comportamento de frameworks de procesamento Big Data como Hadoop, Spark e Flink, moi populares na actualidade. BDEv xestiona a súa configuración e despregamento, xerando os conxuntos de datos de entrada e executando cargas de traballo previamente elixidas polo usuario. Durante cada execución, BDEv extrae diversas métricas de avaliación que inclúen rendemento, uso de recursos, eficiencia enerxética e comportamento a nivel de microarquitectura. Doutra banda, Flame-MR permite optimizar o rendemento de aplicacións Hadoop MapReduce. En xeral, o seu deseño baséase nunha arquitectura dirixida por eventos capaz de mellorar a eficiencia dos recursos do sistema mediante o solapamento da computación coas comunicacións. Ademais de reducir o número de copias en memoria que presenta Hadoop, emprega algoritmos eficientes para ordenar e mesturar os datos. Flame-MR substitúe o motor de procesamento de datos MapReduce de xeito totalmente transparente, polo que non é necesario modificar o código de aplicacións xa existentes. A mellora de rendemento de Flame-MR foi avaliada de maneira exhaustiva en sistemas clúster e cloud, executando tanto benchmarks estándar coma aplicacións pertencentes a casos de uso reais. Os resultados amosan unha redución de entre un 40 % e un 90 % do tempo de execución das aplicacións. Esta Tese proporciona aos usuarios e desenvolvedores de Big Data dúas potentes ferramentas para analizar e comprender o comportamento de frameworks de procesamento de datos e reducir o tempo de execución das aplicacións sen necesidade de contar con coñecemento experto para elo.

Resumen

Hoy en día, muchas organizaciones utilizan tecnologías Big Data para extraer información de grandes volúmenes de datos. A medida que el tamaño de estos volúmenes crece, satisfacer las demandas de rendimiento de las aplicaciones de procesamiento de datos masivos se vuelve más difícil. Esta Tesis se centra en evaluar y optimizar estas aplicaciones, presentando dos nuevas herramientas llamadas BDEv y Flame-MR. Por un lado, BDEv analiza el comportamiento de frameworks de procesamiento Big Data como Hadoop, Spark y Flink, muy populares en la actualidad. BDEv gestiona su configuración y despliegue, generando los conjuntos de datos de entrada y ejecutando cargas de trabajo previamente elegidas por el usuario. Durante cada ejecución, BDEv extrae diversas métricas de evaluación que incluyen rendimiento, uso de recursos, eficiencia energética y comportamiento a nivel de microarquitectura. Por otro lado, Flame-MR permite optimizar el rendimiento de aplicaciones Hadoop MapReduce. En general, su diseño se basa en una arquitectura dirigida por eventos capaz de mejorar la eficiencia de los recursos del sistema mediante el solapamiento de la computación con las comunicaciones. Además de reducir el número de copias en memoria que presenta Hadoop, utiliza algoritmos eficientes para ordenar y mezclar los datos. Flame-MR reemplaza el motor de procesamiento de datos MapReduce de manera totalmente transparente, por lo que no se necesita modificar el código de aplicaciones ya existentes. La mejora de rendimiento de Flame-MR ha sido evaluada de manera exhaustiva en sistemas clúster y cloud, ejecutando tanto benchmarks estándar como aplicaciones pertenecientes a casos de uso reales. Los resultados muestran una reducción de entre un 40 % y un 90 % del tiempo de ejecución de las aplicaciones. Esta Tesis proporciona a los usuarios y desarrolladores de Big Data dos potentes herramientas para analizar y comprender el comportamiento de frameworks de procesamiento de datos y reducir el tiempo de ejecución de las aplicaciones sin necesidad de contar con conocimiento experto para ello.

Abstract

Nowadays, Big Data technologies are used by many organizations to extract valuable information from large-scale datasets. As the size of these datasets increases, meeting the huge performance requirements of data processing applications becomes more challenging. This Thesis focuses on evaluating and optimizing these applications by proposing two new tools, namely BDEv and Flame-MR. On the one hand, BDEv allows to thoroughly assess the behavior of widespread Big Data processing frameworks such as Hadoop, Spark and Flink. It manages the configuration and deployment of the frameworks, generating the input datasets and launching the workloads specified by the user. During each workload, it automatically extracts several evaluation metrics that include performance, resource utilization, energy efficiency and microarchitectural behavior. On the other hand, Flame-MR optimizes the performance of existing Hadoop MapReduce applications. Its overall design is based on an event-driven architecture that improves the efficiency of the system resources by pipelining data movements and computation. Moreover, it avoids redundant memory copies present in Hadoop, while also using efficient sort and merge algorithms for data processing. Flame-MR replaces the underlying MapReduce data processing engine in a transparent way and thus the source code of existing applications does not require to be modified. The performance benefits provided by Flame-MR have been thoroughly evaluated on cluster and cloud systems by using both standard benchmarks and real-world applications, showing reductions in execution time that range from 40% to 90%. This Thesis provides Big Data users with powerful tools to analyze and understand the behavior of data processing frameworks and reduce the execution time of the applications without requiring expert knowledge.

Preface

The use of Big Data technologies for large-scale data processing is widely spreading, transforming the way we extract valuable information from large amounts of data. As the size of the input datasets increases, their analysis becomes a challenging task for current computing systems. This situation puts a lot of strain onto the performance and scalability of current data processing frameworks, which in turn demands for new ways of evaluating and optimizing them. The present Thesis, "Evaluation and Optimization of Big Data Processing on High Performance Computing Systems", addresses these issues by designing and implementing new tools that can help developers and users to identify and alleviate existing performance bottlenecks.

Objectives and Work Methodology

The main objectives of this Thesis are listed below, including some key sub-goals that must be met.

1. Development of an automatic tool to evaluate Big Data frameworks.
    - Automatic configuration and deployment of frameworks.
    - Support for several benchmarks with different characterizations.
    - Multi-metric evaluation.
    - Easy collection of results (e.g. automatic graph generation).

2. Design and implementation of a new MapReduce framework for in-memory data processing.
    - Transparent replacement of Hadoop's architecture.
    - Leveraging of system resources (e.g. CPU, memory).
    - Efficient data pipelining.
    - Application compatibility.
3. Thorough performance evaluation of the new framework.
    - Evaluation in cloud and High Performance Computing (HPC) systems.
    - Execution of standard benchmarks and real-world applications.
    - In-depth analysis and characterization of the workloads.

These objectives have been addressed by using a classic work methodology in research and engineering: analysis, design, implementation and evaluation. This methodology has been applied to the first and second objectives, which correspond with the development cycles of the Thesis. To address the first objective, the current state of the art regarding the assessment of data processing frameworks was reviewed, identifying the most common issues encountered when conducting experimental evaluations in Big Data systems. The results from this analysis were used as a basis for the design of the MapReduce Evaluator (MREv) tool, which later evolved into a more comprehensive tool: Big Data Evaluator (BDEv). Once the implementation of BDEv was complete, it was appropriately tested by using multiple frameworks, workloads and systems.

In order to achieve the second objective, a new framework named Flame-MR was developed. Its performance targets were first identified by analyzing the behavior of Hadoop, using the results obtained by BDEv during the first part of the Thesis. Next, Flame-MR was designed in a modular way, implementing and testing the software components using an iterative approach. When the initial functional version of the framework was complete, several memory management techniques were developed for further performance. The final version of Flame-MR was evaluated in HPC and cloud systems to achieve the third objective of the Thesis, executing both standard benchmarks and real-world applications.

Funding and Technical Means

The means that were used to carry out this Thesis have been the following:

- Working material, human and financial support primarily by the Computer Architecture Group of the University of A Coruña, along with Research Fellowships funded by the Ministry of Education, Culture and Sport of Spain (FPU program, ref. FPU14/02805) and by the Galician Government (Xunta de Galicia, ref. ED 481A-2015/189).
- Access to bibliographical material through the library of the University of A Coruña.
- Additional funding through the following research projects:
  - European funding: "Network For Sustainable Ultrascale Computing" (NESUS COST Action ref. IC1305).
  - State funding by the Ministry of Economy and Competitiveness of Spain through the project "New Challenges in High Performance Computing: from Architectures to Applications" (refs. TIN2013-42148-P and TIN2016-75845-P).
  - Regional funding by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Groups (Computer Architecture Group, refs. GRC2013/055 and ED431C 2017/04) and Network of Cloud and Big Data Technologies for HPC (refs. R2014/041 and ED431D R2016/045).
  - Private funding: project "Spark-based Duplicate Reads Removal Tool for Sequencing Studies" funded by the "Microsoft Azure for Research" program (ref. MS-AZR-0036P), and project "High-Performance Computing and Communications in AWS" funded by a research grant of Amazon Web Services (AWS) LLC.
- Access to clusters, supercomputers and cloud computing platforms:
  - *Pluton* cluster (Computer Architecture Group, University of A Coruña, Spain). 16 nodes with 2 octa-core Intel Xeon Sandy Bridge-EP proces-

sors, 64 GB of memory and 1 HDD disk of 1 TB, interconnected via Gigabit Ethernet and InfiniBand FDR.

  ◦ *Grid'5000* infrastructure (Inria, CNRS, RENATER and several French Universities). For the experiments of the Thesis, 33 nodes have been used, each of them with 2 Intel Xeon octa-core Haswell-EP processors, 128 GB of memory and 2 HDD disks of 558 GB, inteconnected via 4 $\times$10 Gbps Ethernet.
  ◦ *Amazon EC2* IaaS cloud platform (Amazon Web Services, AWS). Two instance types have been used: (1) c3.4xlarge, 2 Intel Xeon octa-core Ivy Bridge-EP processors, 30 GB of memory and 2 local SSD disks of 160 GB; and (2) i2.4xlarge, 2 Intel Xeon octa-core Ivy Bridge-EP processors, 122 GB of memory and 4 local SSD disks of 800 GB. These instances are interconnected via Gigabit Ethernet.
  ◦ *Microsoft Azure* IaaS cloud platform (Microsoft Corporation). L16S instances were utilized, which have 1 Intel Xeon hexa-core Haswell-EP processor, 128 GB of memory and 2.7 TB of SSD local storage, interconnected via 4 $\times$10 Gbps Ethernet.

- A three-month research visit to the Université Grenoble Alpes, France, which has allowed the access to the Grid'5000 infrastructure for evaluating the performance benefits of Flame-MR when executing MapReduce queries belonging to the VELaSSCo project. This research visit was funded by the Ministry of Education, Culture and Sport of Spain through a competitive grant within the FPU program (ref. EST16/00076).

Structure of the Thesis

The Thesis is organized as follows:

- Chapter 1 first introduces the main issues that users and developers find when deploying and using popular Big Data processing frameworks. These issues have motivated the research work that is described in the remainder of the Thesis.

- Chapter 2 presents the state of the art regarding the evaluation of Big Data processing frameworks, including an overview of the most relevant frameworks, benchmark suites and performance studies. Next, the BDEv tool is presented by describing its objectives, characteristics and overall design. A practical use case shows different experimental analysis that BDEv allows to conduct.

- Chapter 3 addresses the transparent acceleration of Hadoop MapReduce applications by presenting the design of the Flame-MR framework, detailing the core characteristics of its architecture along with the approaches followed to avoid some performance issues present in Hadoop. After that, this chapter explains several optimizations implemented to improve memory efficiency and reduce disk overheads, analyzing their benefits in terms of performance and resource utilization.

- Chapter 4 conducts a thorough evaluation of Flame-MR by executing several types of workloads including standard benchmarks and real-world MapReduce applications. Several scenarios are considered, using different HPC and cloud systems with distinct hardware and software characteristics, which guarantees the portability of the optimizations proposed in Chapter 3.

- Chapter 5 extracts some final conclusions from the Thesis and discusses future research lines regarding the development of new models and tools for Big Data frameworks.

Main Contributions

The main original contributions derived from the Thesis are the following:

- Development of BDEv to characterize the performance and automatize the evaluation of Big Data frameworks [122, 125].

- Experimental analysis of the behavior of popular Big Data processing frameworks [123, 126, 129].

- Development of Flame-MR to optimize Hadoop workloads in a transparent way, leveraging computational resources of HPC and cloud systems [127, 128].

- Thorough performance evaluation of Flame-MR using standard benchmarks and real-world Hadoop applications [35, 124].

Developed software

The software tools developed in this Thesis are publicly available:

- BDEv: automatic evaluation tool for Big Data processing frameworks. Available at `http://bdev.des.udc.es/`.
- Flame-MR: in-memory MapReduce framework for transparent optimization of Hadoop applications. Available at `http://flamemr.des.udc.es/`.
- MarDRe: MapReduce tool to remove duplicate and near-duplicate DNA reads in large genomic datasets. Available at `http://mardre.des.udc.es/`.

Registered software

Three software products have been registered in the IP registry as outcomes of this Thesis:

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Flame-MR: MapReduce framework for in-memory computing, 2018. Record entry number: pending. Owning entity: Universidade da Coruña. Priority country: Spain.
- R. R. Expósito, J. Veiga, J. González-Domínguez, and J. Touriño. MapReduce Duplicate Removal tool: MarDRe, November 2017. Record entry number: 03/2018/174. Owning entity: Universidade da Coruña. Priority country: Spain.
- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. MapReduce Evaluator: MREv, June 2016. Record entry number: 03/2016/1054. Owning entity: Universidade da Coruña. Priority country: Spain. In exploitation by Torus Software Solutions S.L. through contract INV13317 since 18/12/2017.

Publications from the Thesis

Journal publications

- J. Veiga, R. R. Expósito, B. Raffin, and J. Touriño. Optimization of real-world MapReduce applications with Flame-MR: practical use cases. 2018. (Submitted for journal publication).

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Enhancing in-memory efficiency for MapReduce-based data processing. *Journal of Parallel and Distributed Computing*, 120:323–338, 2018. JCR Q2.

- J. Veiga, J. Enes, R. R. Expósito, and J. Touriño. BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks. *Future Generation Computer Systems*, 86:565–581, 2018. JCR Q1 (first decile).

- R. R. Expósito, J. Veiga, J. González-Domínguez, and J. Touriño. MarDRe: efficient MapReduce-based removal of duplicate DNA reads in the cloud. *Bioinformatics*, 33(17):2762–2764, 2017. JCR Q1 (first decile).

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Flame-MR: an event-driven architecture for MapReduce applications. *Future Generation Computer Systems*, 65:46–56, 2016. JCR Q1 (first decile).

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Analysis and evaluation of MapReduce solutions on an HPC cluster. *Computers & Electrical Engineering*, 50:200–216, 2016. JCR Q3.

International conferences

- J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Touriño. Performance evaluation of Big Data frameworks for large-scale data analytics. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016)*, pages 424–431. Washington, DC, USA, 2016.

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. MREv: an automatic MapReduce Evaluation tool for Big Data workloads. In *International Conference on Computational Science (ICCS'15)*, pages 80–89. Reykjavík, Iceland, 2015.

- J. Veiga, G. L. Taboada, X. C. Pardo, and J. Touriño. The HPS3 service: reduction of cost and transfer time for storing data on clouds. In *16th IEEE International Conference on High Performance Computing and Communications (HPCC'14)*, pages 213–220. Paris, France, 2014.

Book chapters

- J. Veiga, R. R. Expósito, and J. Touriño. Performance evaluation of Big Data analysis. In S. Sakr and A. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–6. Springer International Publishing, Cham, 2018.

Other minor publications

- J. Veiga, R. R. Expósito, and J. Touriño. Flame-MR: transparent performance improvement of Big Data applications. In *Journée des doctorants de l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII)*. Grenoble, France, 2017.

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Performance improvement of MapReduce applications using Flame-MR. In *2nd NESUS Winter School & PhD Symposium 2017*. Vibo Valentia, Italy, 2017.

Contents

1. Introduction **1**

2. BDEv: automating the evaluation of Big Data frameworks **5**

2.1. State of the art in evaluating data processing frameworks 6

2.1.1. Big Data processing frameworks 6

2.1.2. Benchmarking tools 9

2.1.3. Performance studies of Big Data processing frameworks 12

2.1.4. Energy efficiency 13

2.1.5. Microarchitectural characterization 16

2.1.6. Summary 17

2.2. BDEv: goals and design 17

2.2.1. Motivation 17

2.2.2. BDEv characteristics 19

2.2.3. BDEv design 21

2.2.4. Evaluation metrics 28

2.2.5. Operation 31

2.2.6. Targeted use cases 33

2.3. BDEv outcomes 34
2.3.1. Experimental testbed 35
2.3.2. Performance and energy efficiency 38
2.3.3. Power consumption and resource utilization 39
2.3.4. Microarchitecture-level metrics 43
2.4. Conclusions 46

3. Flame-MR: efficient event-driven MapReduce data processing 49
3.1. Background 50
3.2. Related work 52
3.3. Flame-MR design 55
3.3.1. Flame-MR architecture 55
3.3.2. MapReduce operations 60
3.4. Memory management optimizations 67
3.4.1. Garbage collection reduction 69
3.4.2. Buffer type analysis 76
3.4.3. Iterative support 80
3.5. Conclusions 86

4. Experimental analysis of Flame-MR in cluster and cloud platforms 89
4.1. Related work 90
4.2. Performance comparison with Hadoop and Spark in the cloud 91
4.2.1. Comparison with Hadoop 93
4.2.2. Comparison with Spark 97
4.3. Applicability study: optimization of real-world use cases 101

4.3.1. VELaSSCo: data visualization queries 101

4.3.2. CloudRS: error removal in genomic data 108

4.3.3. MarDRe: duplicate read removal in genome sequencing data . 116

4.4. Conclusions 122

5. Conclusions and future work **123**

References **127**

A. Resumen extendido en castellano **145**

List of Tables

2.1. Comparison of batch, stream and hybrid frameworks 10
2.2. Summary of works evaluating Big Data frameworks 18
2.3. Frameworks supported in BDEv 23
2.4. Benchmarks supported in BDEv 25
2.5. Node characteristics of Grid'5000 36
2.6. Configuration of the frameworks in Grid'5000 37
2.7. Benchmark characteristics 37

3.1. Node characteristics of Pluton 68
3.2. Configuration of Flame-MR in Pluton 68
3.3. Sort results for Flame-MR and Flame-MR-GCop ET: Execution Time; GCT: Garbage Collection Time 74
3.4. Sort results for different buffer types in Flame-MR-GCop ET: Execution Time; GCT: Garbage Collection Time 77
3.5. Execution times for PageRank 84

4.1. Node characteristics of Amazon EC2 instances 92
4.2. Configuration of Hadoop and HDFS in Amazon EC2 94
4.3. Configuration of Flame-MR and Spark in Amazon EC2 94

4.4. Node characteristics of Grid'5000 . . . 107

4.5. Configuration of the frameworks in Grid'5000 . . . 107

4.6. Node characteristics of Pluton . . . 113

4.7. Node characteristics of L16S instances in Azure . . . 113

4.8. Configuration of the frameworks in Pluton . . . 114

4.9. Configuration of the frameworks in Azure . . . 114

4.10. Load balancing in MarDRe . . . 121

List of Figures

2.1. BDEv design overview 22

2.2. Overview of the Intel RAPL architecture for a dual-socket system . . 29

2.3. BDEv control flow 32

2.4. Execution time, energy consumption and ED^2P ratio results (lower is better) 38

2.5. Average power consumption and resource utilization per node for TeraSort 41

2.6. Average power consumption and resource utilization per node for K-Means 42

2.7. Average power consumption and resource utilization per node for PageRank 43

2.8. Microarchitecture-level metrics results 45

3.1. Hadoop data flow with multiple map and reduce tasks 51

3.2. High-level architectural overview of Flame-MR O: Operations B: Buffers C: Chunks 56

3.3. Code examples for WordCount map and reduce functions in Flame-MR and Hadoop 61

3.4. Overview of the MapReduce workflow in Flame-MR O: Operations B: Buffers C: Chunks 62

3.5. k-way merge (k=4) C: Chunks K: Keys V: Values p: pointers 65
3.6. DataPool overview in Flame-MR-GCop 70
3.7. Map output example 72
3.8. GCT and memory usage over time for Flame-MR and Flame-MR-GCop 75
3.9. GCT and memory usage over time for the different DataBuffer implementations in Flame-MR-GCop 79
3.10. Short-lived vs long-lived Workers 81
3.11. Data cache example 82
3.12. Resource utilization statistics of Flame-MR, Flame-MR-It-NoCache and Flame-MR-It-Cache 85

4.1. Execution times of Hadoop, Flame-MR and Flame-MR-It in Amazon EC2 95
4.2. Disk utilization of Hadoop and Flame-MR-It for Sort 96
4.3. Memory utilization of Hadoop and Flame-MR-It for Sort 97
4.4. Execution times of Hadoop, Spark and Flame-MR-It in Amazon EC2 98
4.5. CPU utilization and network traffic for PageRank in i2.4xlarge 99
4.6. CPU utilization and network traffic for Connected Components in i2.4xlarge 100
4.7. Data object serialization in Flame-MR 105
4.8. Execution times of VELaSSCo queries with Hadoop and Flame-MR . 108
4.9. Execution times of CloudRS with Hadoop and Flame-MR 115
4.10. Load balancing mode in Flame-MR version 1.1 119
4.11. Execution times of MarDRe with Hadoop, Flame-MR and Flame-MR-LB 120

Chapter 1

Introduction

The data we generate and collect increases day by day in an exponential fashion. In fact, the International Data Corporation (IDC) forecasts that we will globally generate 163 ZB of data by 2025, ten times the 16.1 ZB generated in 2016 [101]. The term Big Data refers to the use of these huge volumes of data to obtain valuable information, providing solutions to analytical problems in multiple fields such as smart cities [66], social sciences [52], medicine [146], industry [138] and many more. The main challenges that Big Data technologies have to face are defined by Gartner, Inc. as volume, variety and velocity [44]. These three challenges are constantly becoming more difficult to manage, since they put a lot of strain into the performance of data processing systems.

The widespread adoption of Big Data technologies has been encouraged by a change in the way data processing pipelines are defined. The publication of the MapReduce programming model by Google [29] established the basis of a new computing paradigm that focuses on moving the computation to where data is stored rather than moving the data to the computation. As MapReduce processes large datasets distributed among the nodes of a cluster, computational tasks can be collocated within the nodes that contain the input data to be processed. Doing so, it avoids unnecessary and costly data movements through the network. Moreover, MapReduce programmers only need to define the transformation operations that process the data. Low-level implementation details, such as network communications and task parallelism, are hidden to them. Therefore, Big Data developers

can focus on algorithm design, without requiring the programming effort and the training needed by traditional parallel computing paradigms such as the Message Passing Inteface (MPI) [53].

MapReduce is implemented by common Big Data processing frameworks, mainly Apache Hadoop [8]. Other frameworks like Apache Spark [145] or Apache Flink [7] also utilize some of its semantics, although they allow to perform a broader set of data transformations for improved performance and programming productivity. These frameworks process large datasets by executing workloads in cluster and cloud platforms that can have up to thousands of nodes [67]. As the size of the datasets grows, the computing capability required to obtain the demanded information increases, reaching levels that are difficult to handle by traditional systems. This situation requires Big Data developers and users to make a considerable effort to identify and optimize the performance bottlenecks that may be present in current technologies and systems, in order to keep workload execution times within acceptable limits without damaging the quality of the results.

Determining the major factors that limit the performance of Big Data frameworks is far from straightforward, as they are affected by a wide variety of aspects such as their overall design and implementation, their task scheduling mechanism, the workload distribution or the underlying system architecture. This has caused a spur of research activity in assessing the performance and resource utilization of Big Data frameworks. However, the large number of affecting factors makes it difficult to extrapolate the results of these studies to a certain use case prior to obtaining empirical information. Therefore, the identification of the potential performance bottlenecks of a workload requires to perform a thorough experimental analysis. As Big Data users come from many different fields without needing to be aware of the low-level behavior of the frameworks, they might not have the skills and/or the time required to perform these tasks in a fast, effective and accurate way.

Another issue is that Big Data applications are often limited by the performance that the frameworks are able to provide. Clear examples of that can be found in the large amount of Hadoop applications that have been developed since its release in 2007, some of them being the result of months or years of development. Although Hadoop has continued to evolve and improve during this time, some initial design decisions keep hindering its overall performance. In order to alleviate this issue, more

up-to-date frameworks like Spark or Flink have appeared to improve the performance of Hadoop and provide newer and extended APIs. Most workloads that have been ported to these APIs show great performance improvements.

However, a sheer number of applications still use Hadoop for data processing, which reduces the performance that they are able to achieve. Adapting them to Spark or Flink is not always assumable because of the significant programming and testing effort that is required. Moreover, the actual performance benefit that may be obtained is unknown beforehand, as it depends on the characteristics of the workload. This can sometimes cause the performance to be very similar or even worse when executed with Spark or Flink, which would mean a huge waste of time and human resources that could be better employed in other tasks. In some cases, the source code may not even be available, making its modification unfeasible.

The aim of this Thesis is to alleviate the main difficulties of evaluating and optimizing Big Data workloads, providing new tools that can help users to obtain valuable information about the behavior of the frameworks and optimize existing MapReduce workloads without compromising application compatibility. On the one hand, the performance evaluation of the frameworks is addressed by developing a new evaluation tool, Big Data Evaluator (BDEv), which allows to conduct automatic assessments of the most popular frameworks for Big Data processing. This includes the configuration and cluster deployment of such frameworks, the generation of the input datasets, the execution of the workloads and the extraction of several insightful metrics. These metrics are not limited to performance, as they also take into account scalability, resource utilization, energy efficiency and microarchitectural behavior. On the other hand, we also provide transparent performance optimization of existing MapReduce workloads by developing Flame-MR, an in-memory computing framework that redesigns the data processing pipeline of Hadoop. It implements an event-driven architecture that leverages computational resources efficiently, improving the performance of Hadoop applications without changing their source code. The evaluation of Flame-MR performed in public cloud platforms and High Performance Computing (HPC) clusters shows great performance improvements at zero effort and cost for both standard benchmarks and real-world applications.

Chapter 2

BDEv: automating the evaluation of Big Data frameworks

The evaluation of Big Data frameworks is a crucial task to determine their behavior in a certain system, identifying potential performance bottlenecks that may delay the processing of large datasets. While most of the existing works generally focus only on execution time and resource utilization, analyzing other important metrics is key to fully understand the behavior of these frameworks. For example, microarchitecture-level events can bring meaningful insights to characterize the interaction between frameworks and hardware. Moreover, energy consumption is also gaining increasing attention as systems scale to thousands of cores. This chapter discusses the current state of the art in evaluating distributed processing frameworks, while presenting our Big Data Evaluator (BDEv) tool to extract performance, resource utilization, energy efficiency and microarchitecture-level metrics from the execution of representative Big Data workloads. The provided evaluation use cases of BDEv demonstrate its usefulness to bring meaningful information from popular frameworks such as Hadoop, Spark and Flink.

Section 2.1 provides an overview of the state of the art regarding the evaluation of Big Data frameworks, including a classification of methods and tools. Section 2.2 describes the overall design and main targeted use cases of BDEv, along with a description of its behavior. Section 2.3 presents a practical use case of the utilization of BDEv to evaluate Hadoop, Spark and Flink in terms of several evaluation met-

rics, analyzing the obtained results. Finally, Section 2.4 extracts some conclusions about the great versatility that BDEv provides, which eases the optimization and evaluation tasks of developers and users.

2.1. State of the art in evaluating data processing frameworks

This section aims to provide an overview of Big Data processing systems and how they are currently being evaluated. Once the most popular distributed processing frameworks (Section 2.1.1) and benchmark suites (Section 2.1.2) are introduced, several previous works that analyze the performance characteristics of Big Data workloads are presented (Section 2.1.3), including interesting metrics such as their energy efficiency (Section 2.1.4) and microarchitectural performance (Section 2.1.5).

2.1.1. Big Data processing frameworks

Hadoop [8], along with its implementation of the MapReduce model [29], has long been one of the most popular frameworks for large-scale batch processing. Nowadays, recent requirements from the users have made necessary the development of new paradigms, technologies and tools. A clear example of this issue is the appearance of use cases that require iterative and/or stream processing, implying the use of more advanced frameworks. This is mandatory in order to build pipelines that handle and process data arriving in a real-time manner from different sources, which cannot be done with traditional batch processing frameworks such as Hadoop. In general, current Big Data processing systems can be classified in three groups: (1) batch-only, (2) stream-only and (3) hybrid, according to their underlying data processing engine, framework topology and targeted use case.

Batch-only frameworks were the first to appear in order to handle big datasets in a scalable and easy-to-program way. According to the MapReduce paradigm as originally conceived by Google [29], input data is split and processed in chunks following no particular order and generally with no time constraints. This model computes the output data by using two phases: Map and Reduce. The Map phase

extracts the relevant attributes for the computation and the Reduce phase operates them to get the final result. Currently, the most popular batch processing framework is Hadoop MapReduce, together with the Hadoop Distributed File System (HDFS) [108] to manage distributed data storage and Yet Another Resource Negotiator (YARN) [121] for resource management. In addition to running MapReduce jobs, the vast Hadoop ecosystem has become the most commonly used platform to solve Big Data problems, including multiple open-source projects such as the machine learning library Mahout [10], the graph processing engine Giraph [12], the HDFS-based database HBase [9] and many more.

It is worth mentioning several modifications of Hadoop that adapt it to specific interconnection networks such as RDMA-Hadoop [132], or that seek overall performance improvements like NativeTask [141]. RDMA-Hadoop adapts several Hadoop subsystems (e.g. HDFS) to use Remote Direct Memory Access (RDMA) networks like InfiniBand [61], in order to achieve better communication efficiency (e.g. HDFS replication, MapReduce data shuffling). In the case of NativeTask, it presents a novel C++ implementation of the MapReduce model that includes efficient memory allocation and sorting mechanisms.

The second group of Big Data frameworks, stream-only, were developed when the need to process large-sized data streams arose. This is a scenario where batch processing is not applicable due to time constraints, the possibility of having an unending stream and the lack of real-time support. Examples of stream processing frameworks are Storm [62], Heron [73] and Samza [93]. These frameworks follow a different approach than MapReduce, creating a graph-based architecture using pipelines and Direct Acyclic Graphs (DAGs). Data management in streaming frameworks is also different from the batch-only approach, which mainly relies on HDFS. The streaming paradigm introduces the idea of sources and sinks. A source is defined as the origin of the data into the streaming architecture, while the sink is the end where output data is persisted. Although HDFS can still be used, higher-level storage solutions are preferred. Examples of such solutions are queue systems like Kafka [72] or databases like Cassandra [6], which adapt better to the flowing nature of data streams. Moreover, in order to deploy a stream processing architecture, another component is needed to properly manage the data throughout the data flow. To play such a role, a message-oriented middleware is required, such

as the aforementioned Kafka, RabbitMQ [102] or ZeroMQ [59]. Streaming frameworks may also use other resource management tools apart from YARN, especially in cases where the deployed streaming pipelines and architectures need to be running continuously (i.e. if any part of the framework fails, it has to be relaunched in an automatic way). Examples of such tools are Mesos [58] and Aurora [5], used by Heron, while Samza relies solely on YARN. Regarding Storm, it can be integrated with YARN, Mesos and Docker [87], or run stand-alone.

Finally, hybrid solutions such as Spark [145], Flink [7] or Apex [4] try to offer a unified solution for data processing by covering both the batch and stream processing scenarios. These solutions inherit the functionalities offered by batch processing models like MapReduce, as well as the new features of streaming frameworks. To provide a more efficient data processing but remaining usable for stream processing, these solutions follow the DAG design philosophy, while also implementing new architectures with further optimizations. Spark provides a batch processing engine based on a novel data structure, Resilient Distributed Datasets (RDDs) [144], which are in-memory data collections partitioned over the nodes of a cluster. As RDDs keep data in memory, Spark can avoid disk traffic and alleviate some of the issues that hinder the performance of Hadoop, especially for iterative workloads. Spark also provides stream processing capabilities through Spark Streaming [112], which implements a micro-batch processing model by buffering the stream in sub-second increments which are sent as small fixed datasets for batch processing. Moreover, Spark includes other built-in libraries like MLlib [111] for machine learning and GraphX [110] for graph algorithms. Regarding Flink, it relies on the custom-developed Stratosphere platform [2] to specifically target stream processing. Flink defines streams and transformations as the data sources and operations, respectively. Unlike Spark, Flink provides a stream engine that allows handling incoming data on an item-by-item basis as a true stream. It also supports batch processing by simply considering batches to be data streams with finite boundaries. Like Spark, Flink also includes built-in libraries that support machine learning algorithms (FlinkML [43]) and graph processing (Gelly [42]). Apex has been recently released as a new proposal that aims to offer a mature platform that unifies batch and stream workloads. It provides developers with several libraries and tools in order to lower the barrier to entry and support a broad spectrum of data sources and sinks.

As a summary, Table 2.1 shows the main characteristics of the most relevant frameworks discussed in this section.

2.1.2. Benchmarking tools

This section offers an overview of existing benchmarking tools for evaluating data processing frameworks. The most long-lived projects were originally designed for analyzing the performance of batch-only workloads on Hadoop. That is the case of HiBench [60], a popular benchmark suite that supports 19 workloads in its current version (7.0), including micro-benchmarks, machine learning algorithms, SQL queries, web search engines, graph benchmarks and streaming workloads. Apart from Hadoop, it also supports hybrid and stream-only frameworks like Spark, Flink and Storm and message queuing systems like Kafka. However, not all the workloads are available for all the frameworks. HiBench generates the input datasets needed for the workloads and reports the execution time, throughput and system resource utilization as main metrics. Another well-known project, BigDataBench [130], improves the quality of the input data by providing means to generate them from 13 real-world datasets. Furthermore, it supports 47 workloads classified in 7 different types: artificial intelligence, online service, offline analytics, graph analytics, data warehouse, NoSQL and streaming. From version 2.0 on, BigDataBench also includes DCBench [64] and CloudRank-D [80], other benchmark suites which were previously independent. MRBS [105] is a suite oriented to multi-criteria analysis as it takes into account different metrics like latency, throughput and cost. MRBS includes 32 MapReduce workloads from 5 application domains: recommendation systems, business intelligence, bioinformatics, text processing and data mining. Moreover, MRBS can automatically set up the Hadoop cluster on which the benchmark will run using a public cloud provider configured by the user. Once the cluster is running, MRBS injects the dataset and runs the workload, releasing the resources when the experiment concludes. Apart from evaluating the execution time of the workloads, users can also assess the multi-tenancy of a Hadoop cluster by using GridMix [51]. This benchmark launches several synthetic jobs which emulate different users and queues, being able to evaluate Hadoop features like the distributed cache load, data compression/decompression and jobs with high memory requirements or resource utilization (e.g. CPU, disk).

Table 2.1: Comparison of batch, stream and hybrid frameworks

	Paradigm	**Resource manager**	**Data management**	**Real-time**	**Use case**
Hadoop	batch-only	YARN	distrib. filesystems (e.g. HDFS), object storage (e.g. S3)	no	batch processing of non time-sensitive workloads
RDMA-Hadoop	batch-only	YARN	distrib. filesystems (e.g. HDFS), parallel filesystems (e.g. Lustre), object storage (e.g. S3)	no	native support for RDMA networks
NativeTask	batch-only	YARN	distrib. filesystems (e.g. HDFS), object storage (e.g. S3)	no	native optimization for MapReduce workloads
Storm	stream-only	YARN, stand-alone	databases (e.g. Cassandra), queue systems (e.g. Kafka)	yes	low-latency and real-time processing pipelines
Heron	stream-only	YARN, Mesos, Docker, stand-alone	databases (e.g. Cassandra), queue systems (e.g. Kafka)	yes	improvements over Storm
Samza	stream-only	YARN	queue systems (e.g. Kafka)	yes	large data flows accentuating reliability and statefulness
Spark	hybrid	YARN, Mesos, stand-alone	distrib. filesystems (e.g. HDFS), databases (e.g. Cassandra), object storage (e.g. S3)	near real-time	batch and micro-batch processing with streaming support
Flink	hybrid	YARN, Mesos, stand-alone	distrib. filesystems (e.g. HDFS), databases (e.g. Cassandra), queue systems (e.g. Kafka)	yes	stream processing with support for traditional batch workloads
Apex	hybrid	YARN	distrib. filesystems (e.g. HDFS), databases (e.g. Cassandra), object storage (e.g. S3), queue systems (e.g. Kafka)	yes	unified stream and batch processing

Benchmarking tools also exist which enable users to evaluate other Big Data systems built on top of Hadoop. That is the case of PigMix [96], which evaluates Pig [11], a high-level language for expressing data analytics workloads on top of Hadoop. Furthermore, some Big Data benchmarks focus on evaluating the adaptability of Hadoop to traditional database use cases. One example is MRBench [70], which implements 22 relational SQL queries (e.g. select, join). The authors of MRBench describe how these queries can be translated into MapReduce jobs, and the issues that may arise. BigBench [45] proposes a standard benchmark for Big Data that covers a representative number of application profiles. It includes a data model that represents the typical characteristics of Big Data systems (i.e. variety, velocity and volume), and a synthetic data generator that adopts some of its parts from traditional database benchmarks to support structured, semi-structured and unstructured data.

In the last years, new benchmark suites specifically oriented to in-memory processing frameworks have appeared, like SparkBench [75]. It includes 10 workloads with typical usage patterns of Spark: machine learning, graph processing, stream computations and SQL query processing. It takes into account different metrics like execution time, data process rate, shuffle data size, resource consumption and input/output data size. Although little work can be found regarding benchmarks specifically oriented to Flink, some proposals adapt existing ones to its new programming paradigm. That is the case of [15], which uses BigBench to compare Flink and Hive [119], showing that the former can achieve time savings of about 80%.

Not all Big Data benchmarking tools are focused on evaluating data processing systems. For example, the AMPLab benchmark [3] is focused on evaluating data warehousing solutions such as Hive, Tez [103], Shark [136] and Impala [71]. AMPLab uses HiBench to generate the data and performs the evaluation by means of a benchmark that includes scans, aggregations, joins and user-defined functions. The Yahoo! Cloud Serving Benchmark (YCSB) [26] aims to evaluate different NoSQL databases like HBase [9], Cassandra [6], MongoDB [90], Redis [100], Memcached [41], and many others. YCSB currently includes 6 different workloads, providing an input dataset generator and a configurable workload executor.

2.1.3. Performance studies of Big Data processing frameworks

The Hadoop framework has dominated the world of Big Data over the last decade, and thus its performance has been thoroughly addressed by a wide range of papers [30, 37, 38, 126]. However, recent works focus on in-memory processing frameworks due to the better flexibility and performance they provide. That is the reason why Spark is compared with Hadoop in [106], taking into account performance and resource utilization. The results show that Spark can reduce the execution time by 60% and 80% for CPU-bound and iterative benchmarks, respectively. However, Hadoop is 50% faster than Spark for I/O-bound benchmarks such as Sort. Meanwhile, another work [63] claims that frameworks like Twister [34] or parallel paradigms like MPI can provide better performance than Spark for iterative algorithms. By comparing Hadoop, HaLoop [19], Twister, Spark and an MPI library, the authors conclude that Hadoop obtains the worst performance results. Although Spark does not provide the best performance according to [63], it proves to be the most appropriate option for developing Big Data algorithms in a flexible way. This is because Twister does not support HDFS, which is indispensable for storing big datasets, whereas MPI is not a feasible option for developing and maintaining Big Data applications as it does not abstract data distribution, task parallelization and inter-process communications. A similar conclusion is reached in [48], which compares the performance of Spark and MPI for evolutionary algorithms.

Nowadays, Flink attracts increasing interest when evaluating Big Data frameworks, usually being compared with Spark. All the works that compare Flink with Spark conclude that the performance they provide is highly dependent on the workload executed. That is the case of [109], which compares Spark and Flink using standard benchmarks like WordCount, K-Means, PageRank and relational queries. The results show that Flink outperforms Spark except in the case of the most computationally intensive workloads (e.g. WordCount). Another work [16] analyzes the performance of Flink and Spark, configuring Spark both with the default and the optimized Kryo serializers. This work uses three different genomic applications for evaluating the frameworks: Histogram, Map and Join. Flink shows better performance in Histogram and Map, while Spark gets the best results for Join.

In order to provide some insight into the differences between Spark and Flink,

their internal design characteristics are addressed in [82], identifying a set of configuration parameters that have a major influence on the execution time and scalability of these frameworks: task parallelism, shuffle configuration, memory management and data serialization. The benchmarks are also analyzed to identify the data operators they use. The main conclusion is that Spark is 40% faster than Flink for large-graph processing, while Flink is 33% faster than Spark for single-iteration and small-graph workloads. Further evaluations are conducted in [123] but using updated versions of these frameworks, showing that Spark provides better results and stability in general. However, some new features introduced by Flink can accelerate iterative algorithms, like the use of delta iterations in PageRank, which allows reducing the execution time by 70% compared with Spark. The authors also take into account other important parameters of the system: HDFS block size, input data size, interconnection network and thread configuration.

As can be seen, the evaluation of Spark and Flink is gaining attention, not only in terms of performance but also taking into account usability, configuration parameters and resource utilization. The performance obtained by these frameworks is highly dependent not only on the characteristics of the workload, but also on the particular version being evaluated (both are active projects that are continuously evolving). Furthermore, the suitability of the workloads that are usually executed in these works has been discussed in [17], proposing a new evaluation methodology that takes into account the input data size and the characteristics of the data model. Note that all the previous works have focused their evaluations on the batch processing capabilities of Spark and Flink. Other recent works have also assessed their stream processing capabilities [24, 98, 104], comparing them with other stream-only technologies such as Storm or Samza.

2.1.4. Energy efficiency

The energy efficiency of Big Data frameworks has been addressed by previous works under different points of view, studying the main factors that can impact energy consumption and, more recently, developing new proposals in order to decrease it. These works can be classified into three different groups, depending on the method used to get the energy measurements.

The first group is composed by works that estimate the power values by using an energy model. These models usually take into account the power specifications of the underlying node and the utilization of system resources like the CPU. One clear example is [91], an evaluation performed in the Microsoft Azure cloud [88] that uses a model based on the CPU load of the virtual machines to estimate power consumption. The results, which include experiments with a Hadoop cluster, show that heterogeneity of cloud instances harms energy efficiency. This problem is addressed in [23] by developing a new self-adaptive task assignment approach that uses an ant colony algorithm to improve the performance and energy efficiency of MapReduce jobs in heterogeneous Hadoop clusters. The authors modify Hadoop to implement a new scheduling algorithm, obtaining 17% of energy savings compared to the default scheduler. The power measurements obtained in [23] are estimated by using a simple model based on CPU resource utilization and the power consumption of the machine in idle state. More examples of power estimation techniques are included in [89], a survey of different power consumption models for CPUs, virtual machines and servers.

In the second group, power values are obtained by means of an external power meter that is directly connected to the nodes. This is the case of [39], which analyzes the performance and power consumption of several deployment configurations of a Hadoop cluster. The results show that separating data and compute services involves lower energy efficiency than collocating them, and that the power consumption profiles are heavily application-specific. In the experiments, the power metrics were provided by APC Power Distribution Units (PDUs). A similar PDU is used to demonstrate that the energy-aware MapReduce scheduling algorithm proposed in [85] can consume 40% less energy on average. Another work that analyzes the energy efficiency of Hadoop [40] uses a power meter to measure the power consumption of the whole system. The paper identifies four factors that affect the energy efficiency of Hadoop: CPU intensiveness, I/O intensiveness, HDFS replication factor and HDFS block size, giving recommendations related to each of them. Another performance study [76] compares the use of Hadoop on "small" ARM nodes with "big" Intel Xeon ones, concluding that I/O-intensive workloads are more energy efficient on Xeon nodes, while CPU-intensive ones are more efficient on ARM nodes. In this work, power values are recorded by using a Yokogawa power monitor connected to the main electric input line of the system. In [81], "big" Intel Xeon nodes

are compared with “small” Intel Atom ones using a Watts Up Pro power meter. The results show that Xeon nodes perform more efficiently as the input data size increases. The energy consumption of mobile devices can also be measured by using power monitors. In [14], which analyzes the energy efficiency of Big Data stream mobile applications, the batteries are sampled by using a power monitor to measure the energy consumed during 3G/WiFi communications.

The last group of works uses a software interface to access energy counters provided by some CPU vendors. Some of these counters can be accessed by monitoring tools like the Intel data center manager, used in [120] to analyze the energy efficiency of Hadoop on an HPC cluster. This work also proposes the use of the ED^2P metric [84] to evaluate the performance-energy efficiency of Hadoop. Vendor-specific technologies like the HPE integrated Lights-Out (iLO), consisting of a Baseboard Management Controller (BMC) accessible through a REST interface, also allow obtaining power measurements of the node. HPE iLO has been used in [126] to analyze the energy efficiency of different flavors of Hadoop on an HPC cluster, concluding that accelerating the completion of the workloads by using faster interconnects (e.g. InfiniBand) or disks (e.g. SSD) can significantly reduce the energy consumed. However, the most popular way of accessing these energy counters is using power management interfaces that are provided by CPU vendors, which can be used to monitor power in a wide range of modern CPUs. While AMD delivers the Application Power Management (APM) interface, Intel provides the Running Average Power Limit (RAPL) interface [27]. The accuracy of RAPL has been tested in [31], proving that the values it provides can be very useful to characterize the power consumption of an application. RAPL has also been used in [69] to evaluate the energy efficiency of graph processing engines such as Giraph and the Spark GraphX library. This work shows that GraphX is able to consume 42% less energy than Giraph thanks to the use of in-memory RDDs, although it suffers from memory problems that do not appear in Giraph. In [142], RAPL is also used to get power consumption values to compare the horizontal and vertical scalability of a Spark cluster, showing that vertical scalability provides better performance per watt.

2.1.5. Microarchitectural characterization

Most of the previously commented works generally focus on execution time and resource utilization as the only metrics for analyzing performance, while only some of them also take into account energy efficiency. However, there are few works that try to further explore the results obtained in their evaluations by considering other important factors. One interesting example of such metrics is the evaluation of Big Data systems in terms of their microarchitectural performance, by collecting the hardware counters provided by modern CPUs. For instance, available counters allow users to obtain the number of CPU cycles, cache references and branch mispredictions. Note that depending on the CPU model there are different kinds of counters, even across a same vendor (i.e. the availability of these counters is highly CPU-dependent).

In [137], the characterization of Big Data benchmarks aims to identify redundancies in benchmark suites, selecting some representative subsets of HiBench and BigDataBench workloads in order to avoid repetitive results. To do so, the authors execute several benchmarks with Hadoop calculating instructions per cycle, cache miss and branch misprediction ratios, and off-chip bandwidth utilization using the Oprofile tool [94]. Then, they perform a principal component analysis and a hierarchical clustering algorithm to determine which benchmarks are redundant. A similar study is performed in [65], but widening the range of microarchitectural-level metrics that are analyzed and also using other frameworks apart from Hadoop like Spark, Hive and Shark. In this case, the benchmarks are selected from BigDataBench and the metrics are collected using Perf [133]. The work [81] cited in the previous subsection also uses performance counters to compare Intel Xeon and Intel Atom nodes, obtaining the values by means of the Intel VTune performance profiling tool.

Nowadays, the increasing use of memory intensive data analytics is motivating the appearance of new studies that characterize the performance of in-memory frameworks. Intel VTune is employed in [13] to study the CPU and memory intensiveness of several Spark workloads, revealing that the latency of memory accesses is the main performance bottleneck. Another work [143] proposes the effective cache hit ratio, which aims to be more representative than the cache hit ratio when ex-

plaining the relationship between the number of cache hits and the execution times in Spark. The effective cache hit ratio only takes a reference to a cache line as a hit when the dependencies of such line are also located in the cache. Moreover, the authors demonstrate the relevance of their proposal by implementing a new Spark memory manager that handles cache lines and their dependencies as blocks. The results show that their approach speeds up data-parallel jobs by up to 37%.

2.1.6. Summary

This section has provided an in-depth survey regarding the state of the art in benchmarking Big Data processing frameworks by presenting around 50 works that address this topic. As a summary, Table 2.2 includes a group of selected works according to their relevance. The table shows the metrics evaluated in each work: performance, resource utilization, energy efficiency and microarchitectural characterization. It also includes which Big Data frameworks are evaluated and their version (if indicated). The last column shows if there is any publicly available tool to perform the experiments. Finally, the last row includes the metrics and frameworks supported by our tool BDEv in order to provide a direct comparison with previous works.

2.2. BDEv: goals and design

This section first discusses the need for a new tool to carry out in-depth evaluations of Big Data frameworks on a certain system. Next, it describes our proposed Big Data Evaluator (BDEv) tool [122], providing detailed information about its features and design.

2.2.1. Motivation

The selection of a Big Data framework to use in a given system can be affected by several factors. First, the performance of a framework is limited by its underlying design characteristics, depending on aspects like the scheduling of computational tasks

Table 2.2: Summary of works evaluating Big Data frameworks

	Evaluated metrics				**Frameworks**			
Work	**Performance**	**Resources**	**Energy**	**Microarch.**	**Hadoop**	**Spark**	**Flink**	**Avail.**
[106]	✓	✓			✓(2.4.0)	✓(1.3.0)		
[16]	✓					✓(1.3.1)	✓(0.9.0)	
[82]	✓	✓				✓(1.5.3)	✓(0.10.2)	
[123]	✓				✓(2.7.2)	✓(1.6.1)	✓(1.0.2)	✓
[91]			✓(Model)		✓			
[39]	✓		✓(PDUs)		✓(0.20)			
[69]			✓(RAPL)		✓(0.20)	✓(1.4.1)		
[137]				✓(Oprofile)	✓(1.0.3)			
[65]				✓(Perf)	✓(1.0.2)	✓(0.8.1)		
[13]	✓	✓		✓(VTune)		✓		
BDEv	✓	✓	✓(RAPL)	✓(Oprofile)	✓(2.7.3)	✓(2.2.0)	✓(1.3.2)	✓

or the pipelining of CPU and I/O operations. Second, the adaptability of the framework to the system impacts the leveraging of system resources like CPU, memory or network. Moreover, some cluster resources may have to be shared among different applications running in the system, which limits the number of nodes available to use. In those cases, the evaluation must consider different cluster sizes to determine the best balance between performance and cluster occupation. In some environments, specially adapted frameworks can take advantage of specific resources, such as HPC networks like InfiniBand or SSD-based disk technologies. These frameworks must be carefully configured to ensure the fairness of the results.

Big Data users must be aware of all these issues when selecting an available framework, understanding how it works, learning to configure it properly and checking its correct operation. Moreover, the user has to elaborate a set of workloads, searching for appropriate implementations for each framework and ensuring a fair comparison between them. The evaluation process for each framework involves the configuration and deployment of its daemons over the cluster and the execution of the benchmarks. Once the evaluation is finished, the user must also access the output files to check the successful completion of the benchmarks, copying the desired data to a separate file to generate the output graphs.

All these issues turn the evaluation of Big Data frameworks into a difficult and tedious task. Our proposal to overcome them is BDEv, an evaluation tool that enables in-depth, automatic assessment of Big Data frameworks. It includes different representative workloads, unifying their configuration and generating user-friendly information and reports. The design of BDEv is based on our previously released tool, MapReduce Evaluator (MREv) [125], which was only focused on MapReduce frameworks. Currently, BDEv is aimed at two main goals: (1) comparison between frameworks in terms of performance and scalability; and (2) multi-evaluation in terms of several factors like energy efficiency, resource utilization and microarchitectural-level metrics.

2.2.2. BDEv characteristics

BDEv is based on the following features:

- **Unified configuration**

  The evaluation parameters utilized in BDEv help to homogenize the configuration of different frameworks. By allocating the same amount of system resources to each framework, users can expect a fairer comparison between them.

- **Automation of experiments**

  BDEv is able to carry out the experiments without any interaction from the user. Once the evaluation parameters are defined, BDEv performs the entire experiment cycle in an automatic way, including the setting of the frameworks, the generation of the input datasets and the execution of the workloads over the cluster.

- **Leveraging of system resources**

  The configuration of the frameworks is automatically set by detecting the resources available in the system, like the number of CPU cores or the memory size. Nevertheless, users can change any of these parameters to fit their specific needs. BDEv also allows users to configure the frameworks to take advantage of resources that are typically available in HPC systems, like the IP over InfiniBand (IPoIB) interface.

- **Multi-metric evaluation**

  The outcome of the experiments includes the output and execution time of the workloads, along with useful statistics related with resource utilization (e.g. CPU, disk, network), energy efficiency and microarchitectural events. All that information enables the user to analyze the behavior of the frameworks from a holistic point of view.

- **Flexibility**

  BDEv can evaluate Big Data systems in different ways, adapting itself to the particular needs of the user. Therefore, it provides a wide set of experimental options that can be configured to determine the aspects (e.g. configuration parameters, frameworks, workloads) that are evaluated in each experiment. Note that default, safe values are provided to the user.

- **Portability**

  BDEv aims to be easily executed in different kinds of systems. This involves the use of some system-dependent configuration parameters, which can be defined by the user, as well as the awareness of the environment where the experiments are being run (e.g. automatic integration with job schedulers in HPC systems).

- **Error and timeout detection**

  In some cases, errors or exceptions can occur during the experiments. If they are not detected, they can lead to incorrect measurements. BDEv analyzes the output of the workloads to check for errors, avoiding the use of erroneous executions for the final results. Users can also configure a timeout threshold, so if a workload exceeds this value the execution is aborted and its results are discarded.

- **Easy collection of results**

  The information obtained by the different metrics is analyzed to extract a summary that includes statistical results and automatically generated graphs. This eases the comparisons made by users.

2.2.3. BDEv design

BDEv has been implemented following a modular design where the main functionalities are provided by separate packages, which are shown in Figure 2.1. Each of these packages along with their main components are described next.

Experiment

The *Experiment* package contains the components related to the general behavior of BDEv. The *Workflow manager* determines the operations required to carry out the evaluation by using the experiment parameters provided by the *Configuration manager*. Then, it uses the *Framework launcher* and the *Workload runner* to schedule framework- and workload-related operations, respectively. When the

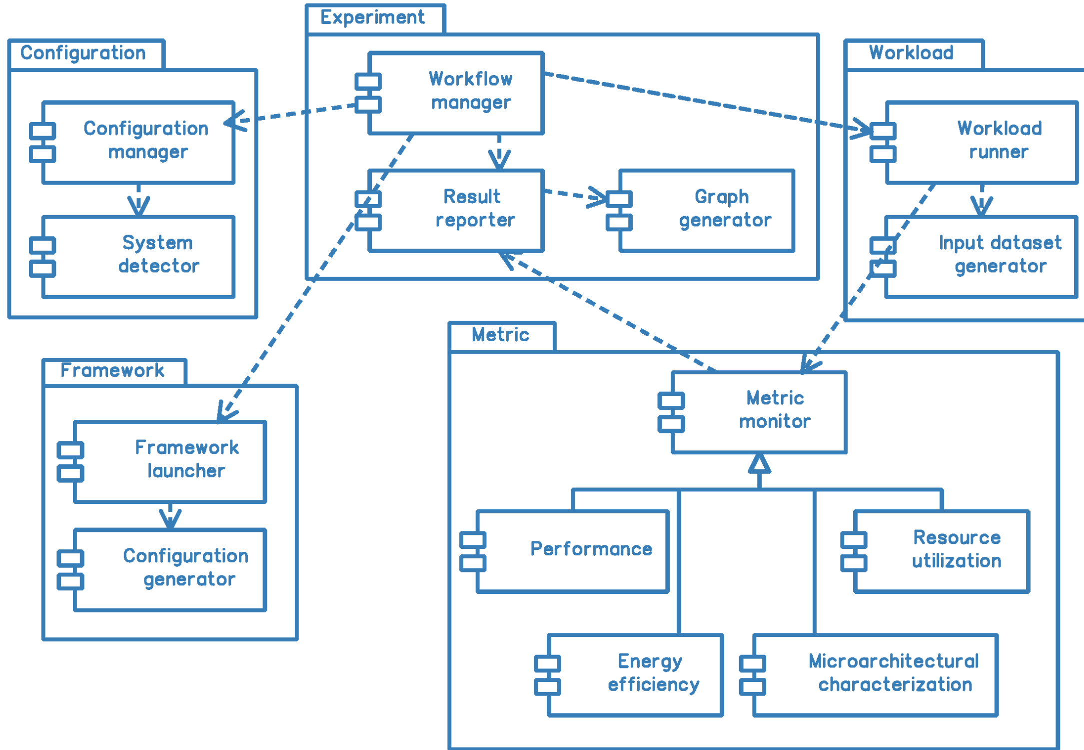


Figure 2.1: BDEv design overview

evaluation finishes, the *Result reporter* creates the summary of the experimental results, generating related graphs by using the *Graph generator*.

Configuration

The *Configuration* package contains the components that determine the parameters to be used in the evaluation. The *Configuration manager* reads the parameters that have been defined by the user in the configuration directory. These parameters are related to different aspects, like system characteristics (e.g. directory to store the temporary files, network interfaces to use), HDFS configuration (e.g. block size) and other framework-specific parameters (e.g. number of workers per node, sort buffer size). They also include the experiment parameters defined by the user, such as the cluster size, frameworks and workloads to be evaluated. If any system-related parameter is not set, the *System detector* determines the default value by analyzing

Table 2.3: Frameworks supported in BDEv

Framework	Version	Network interconnects
Hadoop	1.2.1 / 2.7.6 / 2.8.3 / 2.9.0 / 3.1.0	GbE / IPoIB
Hadoop-UDA	1.2.1 / 2.7.6 / 2.8.3 / 2.9.0 / 3.1.0	IPoIB & RDMA
RDMA-Hadoop	0.9.9	(GbE / IPoIB) & RDMA
RDMA-Hadoop-2	1.2.0 / 1.3.5	(GbE / IPoIB) & RDMA
Spark	1.6.3 / 2.3.0	GbE / IPoIB
RDMA-Spark	0.9.4 / 0.9.5	(GbE / IPoIB) & RDMA
Flink	1.3.2 / 1.4.2	GbE / IPoIB
DataMPI	0.6.0	GbE / IPoIB
Flame-MR	0.10.0 / 1.1	GbE / IPoIB

the system, like the available memory size or the number of CPU cores.

Framework

The components of the *Framework* package control the correct configuration and launching of the frameworks to be evaluated. The *Framework launcher* manages the deployment of the underlying daemons over the cluster that are needed to run each framework (e.g. NameNode/DataNode for HDFS), stopping them once the evaluation has finished. Before the launch, the configuration directory of the framework is set up by using a *Configuration generator* that is specific for each framework and uses the parameters previously defined by the *Configuration manager*.

Table 2.3 contains the frameworks currently supported in BDEv 3.1, their tested versions and the network interconnects they can use. Other versions of the frameworks can also be used, as the support of minor releases is straightforward. However, changing the major release may require some changes in BDEv. Note that at the time of writing the last major versions of Hadoop (3.x) and Spark (2.x) are supported.

Apart from Hadoop, Spark and Flink, BDEv also supports some modifications of these frameworks that use RDMA communications, like Hadoop-UDA [131], RDMA-

Hadoop [132] and RDMA-Spark [79]. Furthermore, other frameworks completely implemented from scratch like DataMPI [78] and Flame-MR [127] are also supported. To our knowledge, no other benchmarking tool provides support for as many frameworks as BDEv.

Workload

The components related to the execution of the workloads are contained in the *Workload* package. First, the *Input dataset generator* builds up the datasets required for their execution. Next, the *Workload runner* executes the selected workloads, using the *Metric monitor* components to record the different metrics.

BDEv supports different benchmark types: standard micro-benchmarks, graph algorithms, machine learning workloads and SQL queries. It also allows executing interactive and batch user-defined commands. The input data generators are specific to each benchmark type, but the user can also define its own input generator. Table 2.4 contains the benchmarks currently included in BDEv and their corresponding input dataset generators. The table also includes the origin of the source code of the benchmarks, which have been carefully studied in order to provide a fair performance comparison. Hence, each framework uses a benchmark implementation based on the same algorithm, reading the same input and writing the same output from/to HDFS. Although the algorithm remains unchanged, each framework employs an optimized version adapted to its available functionalities. Further details about each benchmark are given next.

- **TestDFSIO**

  Tests the throughput of HDFS by generating a large number of tasks performing reads and writes simultaneously. It has been extracted from the examples ("ex." in the table) provided by the Hadoop distribution.

- **WordCount**

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

Table 2.4: Benchmarks supported in BDEv

Category	Benchmark	Input Generator	Benchmark source		
			Hadoop	**Spark**	**Flink**
Micro-benchmark	TestDFSIO	-	Hadoop ex.	-	-
	WordCount	RandomTextWriter	Hadoop ex.	Adapted from ex.	Adapted from ex.
	Grep		Hadoop ex.	Adapted from ex.	Adapted from ex.
	Sort		Hadoop ex.	Adapted from ex.	Adapted from ex.
	TeraSort	TeraGen	Hadoop ex.	Adapted from [117]	Adapted from [117]
Graph processing	ConComp	DataGen	Pegasus	Graphx	Gelly
	PageRank		Pegasus	Adapted from ex.	Adapted from ex.
Machine Learning	Bayes	DataGen	Mahout	MLlib	-
	K-Means	GenKMeansDataset	Mahout	MLlib	Adapted from ex.
SQL queries	Aggregation	DataGen	Hive	Hive	-
	Join		Hive	Hive	-
	Scan		Hive	Hive	-
User	Command	Provided by the user	-	-	-

- **Grep**

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

- **Sort**

  Sort is an I/O-bound workload that is used to order an input text dataset generated by RandomTextWriter. Hadoop includes it in its distribution, while it has been adapted from the examples for Spark and Flink.

- **TeraSort**

  A standard I/O-bound benchmark that sorts 100 byte-sized key-value tuples. This workload assesses the shuffle and sort capabilities of the frameworks. The reference implementation is provided by Hadoop, along with the corresponding input data generator (TeraGen). Spark and Flink do not provide any official implementation in their distributions. So, we have adapted the implementations provided in [117], which are compliant with the Hadoop one.

- **Connected Components (ConComp)**

  An iterative graph algorithm that calculates the subnets of elements that are interconnected. It is included in Pegasus [68], a graph mining system for Hadoop. Both graph-oriented libraries of Spark and Flink, GraphX and Gelly, respectively, contain an implementation of this algorithm. The input dataset is generated by using the DataGen tool included in HiBench.

- **PageRank**

  An iterative graph algorithm that obtains a ranking of the elements of a graph, taking into account the number and quality of the links to each one. It uses the same input data generator as Connected Components. Pegasus also provides this algorithm for Hadoop, while the implementations for Spark and Flink have been adapted from their examples. Moreover, alternative implementations are supported by using the algorithms available in GraphX and Gelly, although the performance of the examples is better according to our experiments.

- **Bayes**

  An iterative clustering algorithm that classifies an input set of elements by determining their probability to belong to several classes. Hadoop executes this algorithm by using the Mahout project, while Spark uses its built-in machine learning library MLlib. The input dataset is generated by the DataGen tool. Flink neither supports this benchmark in FlinkML nor provides an example, and so it is not currently included in BDEv.

- **K-Means**

  An iterative clustering algorithm that classifies an input set of N samples into K clusters. Mahout provides the implementation for Hadoop and the input data generator (GenKMeansDataset). The Spark implementation is provided by MLlib, but there is no such counterpart in FlinkML. However, in this case Flink provides a code example that has been adapted to be included in BDEv.

- **SQL queries**

  Aggregation, Join and Scan are typical database queries that extract information from a database stored in Hive. The input dataset that these queries process is generated by the DataGen tool. BDEv currently supports these queries for Hadoop and Spark.

Metric

This package contains the monitors that extract the evaluation metrics configured by the user. The *Metric monitor* launches the monitoring processes over the cluster when a workload starts, stopping them when it finishes. Then, it communicates with the *Result reporter* and the *Graph generator* to create the reports and graphs, respectively, associated with the recorded data.

Each subcomponent of the *Metric monitor* specializes on a specific metric. As mentioned in Section 2.2.2, BDEv aims to enable holistic evaluations of Big Data frameworks by providing multiple evaluation metrics. The next section provides an overview of the supported metrics and their implementation.

2.2.4. Evaluation metrics

This section provides an overview of the metrics available in BDEv: performance, resource utilization, energy efficiency and microarchitectural characterization.

Performance

BDEv eases the analysis of the frameworks in terms of performance by measuring the execution time of the workloads. After an experiment is complete, the execution time is stored in a summary file. An associated graph is also generated containing the average, maximum and minimum execution time of each workload. Scalability comparisons are also supported by showing the execution time with each cluster size.

Resource utilization

Resource utilization results are useful to analyze the behavior of the frameworks during an evaluation, identifying potential bottlenecks. To do so, BDEv monitors system resources: CPU, disk, memory and network. When a workload is executed, *Resource utilization* monitors are launched in each node of the cluster as previously described in Section 2.2.3. These monitors make use of the dstat utility [33] to record the results, which allows data extraction of all system resources in real time. After the workload finishes, the monitors are stopped and the results of each individual node are gathered and processed to generate the corresponding graphs. Average values among the nodes are also calculated for each resource type.

Energy efficiency

Section 2.1.4 described how energy efficiency is usually assessed in Big Data evaluations: using an energy model, a physical power meter or a software interface. In BDEv, we have chosen the latter alternative as it provides more accurate power information than using a model, also ensuring the portability across several systems, which is not possible with power meters. We have used the RAPL interface that is available in all Intel processors from the Sandy Bridge microarchitecture onwards.

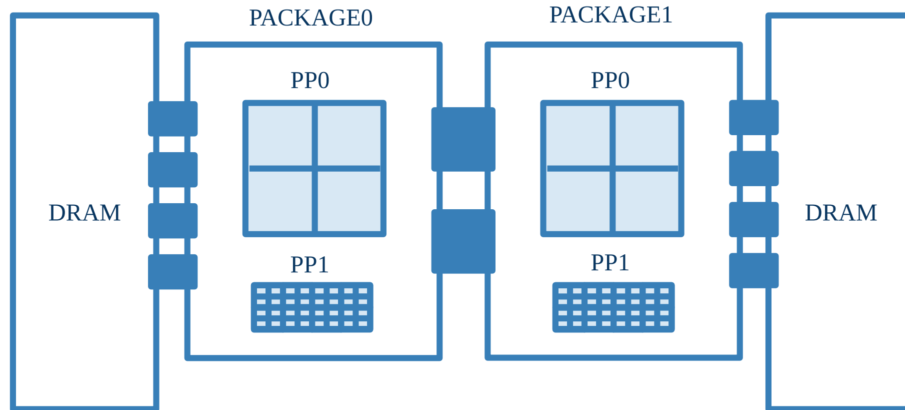


Figure 2.2: Overview of the Intel RAPL architecture for a dual-socket system

We plan to extend this support to AMD processors in the near future by using the APM interface.

Basically, RAPL provides an easy way to keep the power of the processor within a user-defined limit. The internal circuitry of the CPU can estimate the current energy consumption in a highly accurate way, providing these results to the user via Model-Specific Registers (MSRs). These results are directly associated with components of the Intel RAPL architecture, depicted in Figure 2.2. A package identifies the energy consumption measurements of each CPU socket (i.e. each processor). Inside each package, separated energy values are provided for the processing cores, labeled as PP0 (Power Plane 0), and the integrated graphic card (if any), labeled as PP1. These energy values always have the following relationship: PP0 + PP1 <= package. RAPL also provides independent energy values for the memory modules that are associated to each package, labeled as DRAM. Note that the figure shows the architecture of a dual-socket system, each socket providing a quad-core processor and a graphic card. Depending on the particular processor microarchitecture, the measurements of some of these components may not be available.

In order to implement the *Energy efficiency* monitor in BDEv, we have adapted, extended and integrated an existing tool [99] that accesses RAPL counters using the Performance API (PAPI) interface [134]. Amongst many other features, PAPI provides a standard interface for accessing hardware counters and its RAPL component allows us to obtain energy consumption values. The monitor included in BDEv detects all RAPL-related events available in a node and records the energy

values using a configurable time interval, storing them to the corresponding output files. Note that RAPL only provides energy consumption values, and so the power consumption is calculated by BDEv based on the energy consumed in each time interval.

When the execution is finished, the *Graph generator*, using the information elaborated by the *Result reporter*, builds the time series graphs corresponding to the recorded values for each node. The total energy consumed by all the nodes of the cluster and the average power consumption per node are also calculated, and the corresponding graphs are automatically generated for both metrics. Furthermore, additional graphs that show the ED^2P metric, mentioned in Section 2.1.4 and used in the evaluation performed in Section 2.3.2, are also generated.

Microarchitectural characterization

The microarchitectural characterization of Big Data frameworks can provide useful insight on the data operations (e.g. map, join) that can be limiting the performance of the workloads. As mentioned in Section 2.1.5, most current processors provide access to a set of hardware performance counters that allow performing a fine-grained analysis in terms of several microarchitectural characteristics. These counters can detect and count certain microarchitectural events from several hardware sources such as the CPU pipeline or the different cache levels. Such events can help to characterize the interaction between the applications and the hardware, thus providing a more precise picture of the hardware resource utilization. In fact, existing projects like Spark Tungsten [97] are specifically focused on microarchitectural aspects (e.g. cache-aware computation) to improve overall performance. BDEv allows programmers to accelerate this kind of developments by automating the extraction of microarchitectural metrics.

The *Microarchitectural characterization* monitor collects microarchitecture-level performance data about the underlying hardware by keeping count of the events that happen in each node during the execution of a workload. An example of such events is *LLC_MISSES*, which obtains the number of misses in the last level cache. We have chosen Oprofile [94] as our base profiling tool, as it is able to provide hardware counter values for the entire system where it is running. Using such a performance

profiling tool is less intrusive as it does not require manual instrumentation in the source code as needed when using certain APIs. The monitor included in BDEv is easy to configure by indicating the specific events to be recorded. Once the user sets these parameters by modifying the appropriate configuration files, the *Configuration manager* provides the corresponding values to the monitor.

The microarchitectural monitor operates as described in Section 2.2.3, starting the monitoring when a workload begins. The output results include the values of each event counter for each computing node and the total sum of the events occurred in the cluster during the execution. Furthermore, the summary graphs generated at the end of the experiments gather the values related to each event for each framework and workload, easing the comparison between experiments.

2.2.5. Operation

Figure 2.3 shows the high-level control flow of an execution with BDEv. At the beginning of the experiment, BDEv is initialized by setting the configuration parameters specified by the user and creating the output directory. BDEv iterates over the selected cluster sizes and frameworks. Once it configures a framework, it launches the daemons and initializes the benchmarks, generating the required input datasets. Before running a benchmark, its output subdirectory is created and the monitors that collect the metric results are launched. When the benchmark has finished, BDEv automatically generates the metric-related data files and graphs, saving the information of each cluster node and calculating the summary values. Once the benchmark has been executed the number of times configured by the user, the most relevant performance results are appended to the summary report. After executing the selected benchmarks with a particular framework, its daemons are shut down to start executing the next framework (if any).

All the results from an evaluation are stored in an output directory, which includes the summary report, the BDEv log, the output subdirectories for each benchmark and the evaluation graphs. The summary report shows the configuration parameters and a summary of the main performance results. The BDEv log contains the sequence of events during the evaluation, such as the successful configuration and operation of each framework or the end of the execution of each benchmark.

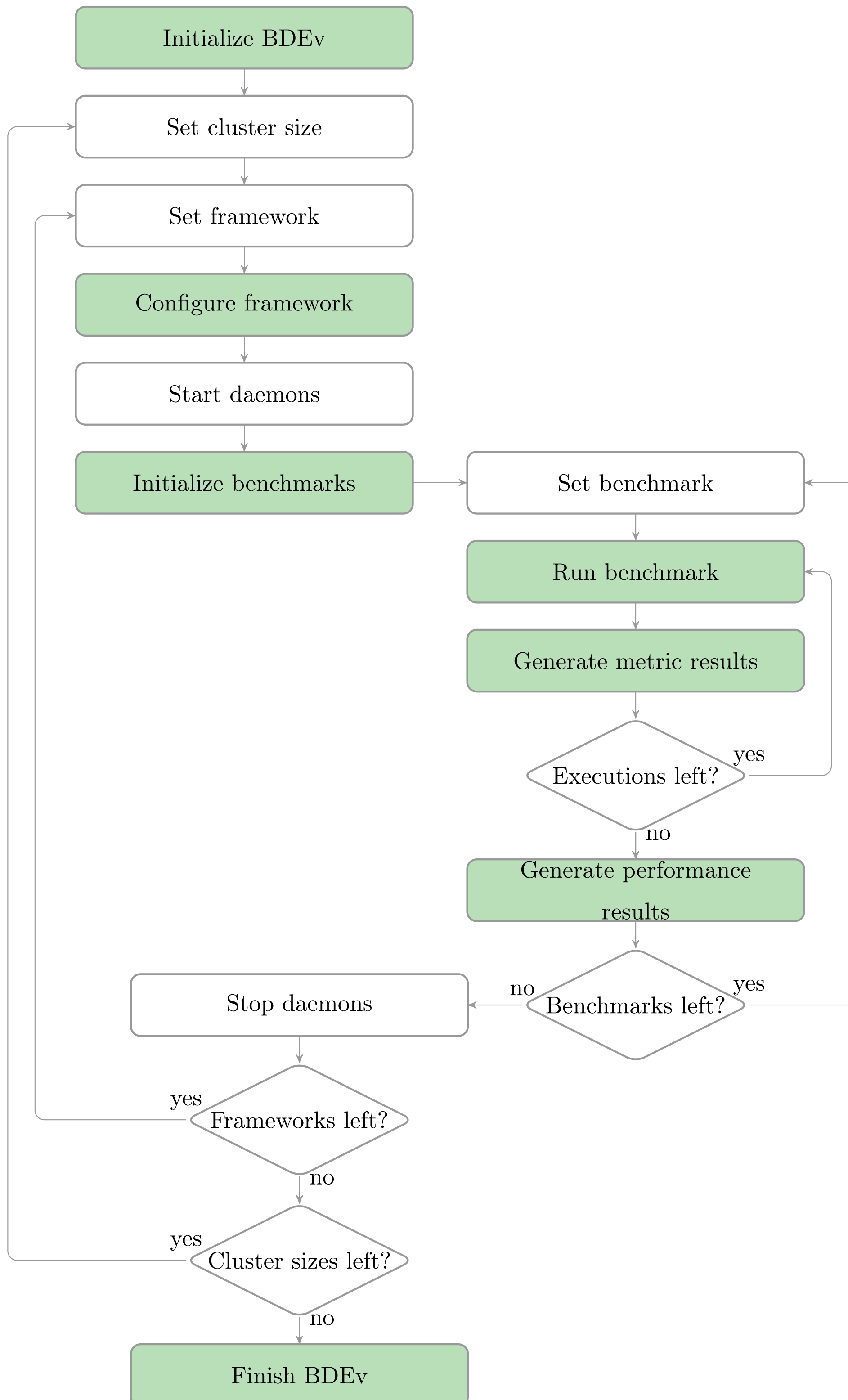


Figure 2.3: BDEv control flow

The output subdirectories of the benchmarks include the standard output, execution times and metrics results for each framework and cluster size, both in text and graphical format. Finally, the evaluation graphs allow to carry out visual comparisons of the frameworks in terms of performance and scalability, resource utilization, energy efficiency and microarchitectural behavior.

2.2.6. Targeted use cases

The use of BDEv is not restricted to a single scenario. Several kinds of users can benefit from its features in a different way, including developers, system administrators and end users. This section provides more details about the usefulness of BDEv in each use case.

- **Developers**

  The development cycle of a Big Data framework or application generally includes several tests that serve to check the correct functioning of the solution. BDEv can automate this task by defining one or more experimental cases that process a certain dataset to get an expected result, detecting errors or timeouts during the execution. Moreover, the standard output of the workload can be analyzed to verify if the obtained result agrees with the expected one. Similarly, sometimes a component of a framework or an application is modified to optimize a certain metric, like the utilization of some resource (e.g. disk, network) or a microarchitectural metric (e.g. data cache locality). Using BDEv, the developer can: (1) identify the component to be optimized by analyzing the results from the evaluation metrics provided by BDEv; (2) once the optimized version has been implemented, BDEv can compare it with previous versions by stressing the targeted component and obtaining new metric results. For instance, developers can analyze the impact of load balancing issues on the power profile of a framework by defining several experiments that present distinct distributions, from balanced to skewed.

- **System administrators**

  The configuration of a Big Data framework for a specific hardware infrastructure involves the definition of a large set of parameters. This can turn

into an overwhelming task, considering that each parameter may have different possibilities that affect the performance of the framework. In some cases, selecting an appropriate setting for a parameter involves an experimental evaluation of the different values. BDEv can ease this task by automatically evaluating those configurations, previously defined by the system administrator. As BDEv configurations can be defined separately (i.e. by using several configuration directories), the total set of possibilities can be established a priori, carrying out the experiments straightforwardly without needing user interaction. Apart from memory and CPU settings, BDEv also considers the configuration of multiple disks and the use of high performance resources like the IPoIB interface, allowing administrators to analyze the impact of their utilization on the different metrics.

- **End users**

  Big Data users often come from many different research fields, sometimes without a thoughtful understanding of the insights of Big Data frameworks or the kind of workloads more suited for each one. Hence, the use of an evaluation tool like BDEv can ease the task of selecting the most suitable implementation of a workload that is available for several Big Data frameworks (e.g. K-Means for Hadoop, Spark and Flink). The selection of the framework will depend on the particular metric (e.g. performance, energy consumption) that the user may want to optimize. BDEv can also benefit those users that need to adjust the configuration parameters of a framework in order to optimize an application according to a certain metric (e.g. adjusting the HDFS block size to improve performance).

2.3. BDEv outcomes

This section presents a practical use case of BDEv by analyzing popular Big Data frameworks taking into account all the evaluation metrics described in Section 2.2.4: performance, energy consumption, resource utilization and microarchitectural characterization. In order to do so, BDEv has been used to deploy several frameworks and execute different standard benchmarks on an HPC cluster, extrac-

ting the metrics in an automatic way. The results demonstrate the potential of our tool to provide meaningful insights from the behavior of Big Data processing frameworks. Section 2.3.1 describes the experimental testbed employed in the evaluation. Section 2.3.2 analyzes the results obtained in terms of the execution time and energy efficiency, while Section 2.3.3 assesses the power consumption and resource utilization. Finally, Section 2.3.4 characterizes the frameworks according to their microarchitecture-level performance.

All the metrics shown in the graphs belong to the same base experiment, which corresponds to the one that obtained the median execution time among a set of 10 measurements. Note that the observed standard deviations were not significant and so they are not shown for clarity purposes. In order to ensure the same conditions for each framework and workload, the operating system buffer cache of the nodes has been erased before executing each experiment.

2.3.1. Experimental testbed

This section first describes the hardware and software characteristics of the system. Then, it details the frameworks and workloads being evaluated.

System configuration

The experiments have been executed in the Grid'5000 infrastructure [50], using a 16-node cluster that corresponds with 1 master and 15 slaves. The main hardware and software characteristics of the nodes are described in Table 2.5. Basically, each node provides 2 Intel Haswell-based processors with 8 physical cores each (i.e. 16 cores per node), 128 GB of memory and 2 local disks of 558 GB each.

Evaluated frameworks

The evaluation has focused on popular state-of-the-art in-memory processing frameworks: Spark and Flink. Hadoop, the most widespread batch processing framework, has also been analyzed to use its results as a baseline. The experiments have been carried out using the last stable versions as of July 2017 (Hadoop

Table 2.5: Node characteristics of Grid'5000

Hardware configuration	
CPU model	2 × Intel Xeon E5-2630 v3 (Haswell)
CPU Speed (Turbo)	2.40 GHz (3.20 GHz)
#Cores	16
Cache (L1 / L2 / L3)	32 KB / 256 KB / 20 MB
Memory	128 GB DDR4 2133 MHz
Disk	2 × 558 GB HDD
Network	4 × 10 Gbps Ethernet
Software configuration	
OS version	Debian Jessie 8.5
Kernel	3.16.0-4
Java	Oracle JDK 1.8.0_121
Scala	2.11.8

2.7.3, Spark 2.2.0 and Flink 1.3.2). Their configuration has been set according to their corresponding user guides and taking into account the characteristics of the underlying system (e.g. number of CPU cores, memory size). The most important parameters of the resulting configurations are shown in Table 2.6. This table also shows the main parameters for HDFS, which has been used to store the input and output datasets.

Workloads

The workloads that have been assessed are common batch processing benchmarks selected to represent different use cases: sorting (TeraSort), machine learning (K-Means), graph analysis (Connected Components) and web search indexing (PageRank), already introduced in Section 2.2.3. The characteristics of the workloads are summarized in Table 2.7, including the size of the input dataset and the data generator. The size of the datasets has been adjusted to keep the execution times into reasonable ranges. Furthermore, the iterative benchmarks have been executed until convergence (i.e. until reaching a final solution).

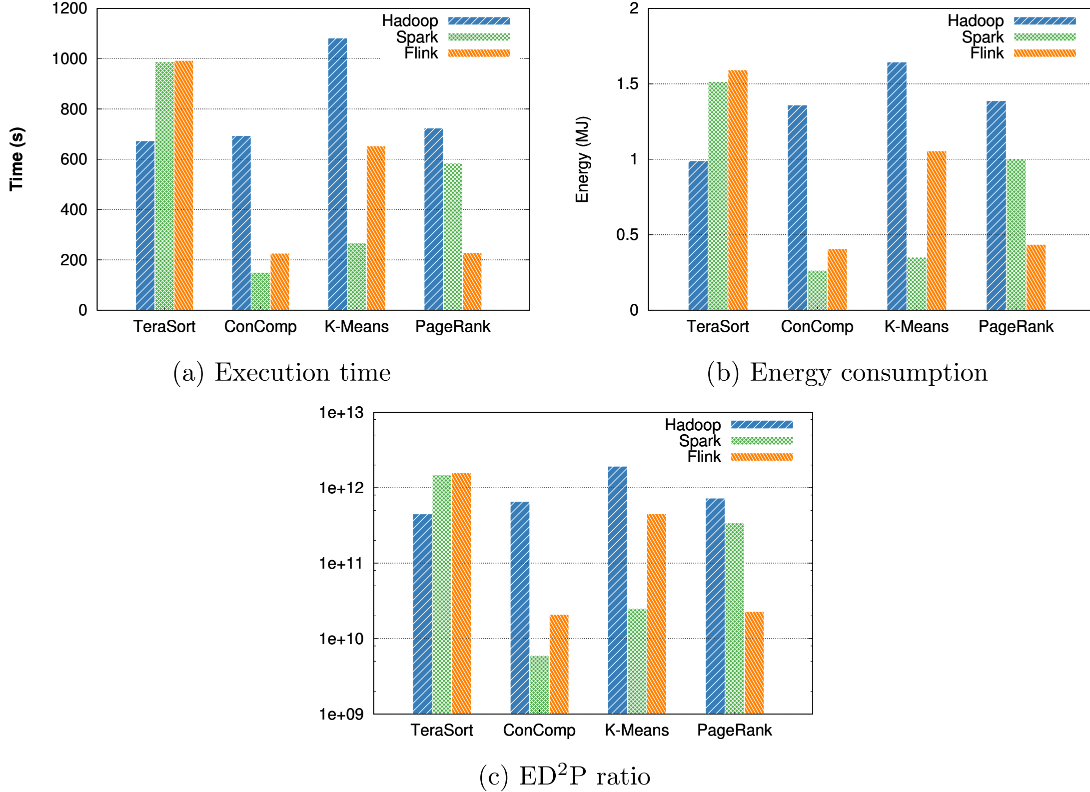
Table 2.6: Configuration of the frameworks in Grid'5000

Hadoop		**HDFS**	
Mapper/Reducer heap size	3.4 GB	HDFS block size	512 MB
Mappers per node	16	Replication factor	1
Reducers per node	16		
Shuffle parallel copies	20		
IO sort MB	852 MB		
IO sort spill percent	80%		

Spark		**Flink**	
Executor heap size	109 GB	TaskManager heap size	109 GB
Workers per node	1	TaskManagers per node	1
Worker cores	32	TaskManager cores	32
Default parallelism	480	Network buffers per node	20480
		Parallelism	480
		IO sort spill percent	80%

Table 2.7: Benchmark characteristics

	Input dataset	
Benchmark	**Size**	**Generator**
TeraSort	300 GB	TeraGen
K-Means	39 GB (N=900M, K=5)	GenKMeansDataset
ConComp	20 GB (30M pages)	DataGen
PageRank	20 GB (30M pages)	DataGen



(a) Execution time

(b) Energy consumption

(c) $\mathrm{ED^2P}$ ratio

Figure 2.4: Execution time, energy consumption and ED^2P ratio results (lower is better)

2.3.2. Performance and energy efficiency

Once the configuration of the experiments has been defined, the experimental results from the evaluation of Hadoop, Spark and Flink with BDEv are presented.

Figure 2.4a shows the performance of the frameworks in terms of execution time, while Figure 2.4b presents their energy efficiency in terms of total energy consumed. Both metrics have been used to calculate the corresponding ED^2P ratio, displayed in Figure 2.4c using a logarithmic scale. The ED^2P metric was first proposed in [84] and measures the performance-energy efficiency of a workload as shown in Equation 2.1.

$$ED^2P = Energy\,Consumed \times (Execution\,Time)^2 \tag{2.1}$$

As can be seen in Figure 2.4a, Hadoop is the best framework for TeraSort, outperforming Spark and Flink by 32%, which shows off the great data sorting capabilities of the MapReduce engine. The energy reductions provided by Hadoop are slightly higher (see Figure 2.4b): 35% and 38% lower energy consumption than Spark and Flink, respectively. This is explained by the fact that Hadoop shows lower CPU utilization than the other frameworks for TeraSort (as will be analyzed in Figures 2.5d, 2.5e and 2.5f), thus leading to lower energy consumption. As expected, in-memory frameworks clearly outperform Hadoop by a large margin when running iterative workloads. For instance, Spark is 34% faster than Flink and 78% faster than Hadoop for Connected Components, providing similar percentages of energy reduction according to Figure 2.4b. Regarding K-Means and PageRank, the best performers are Spark and Flink, respectively. Note also that the ED^2P metric reveals that the best performer for K-Means and PageRank provides a 10x improvement over the second one. This indicates that the results of the best framework are considered far better when energy consumption is taken into account. Finally, it is worth mentioning the variability of the energy consumption for different workloads. While the execution times of Spark and Flink are generally proportional to the energy consumed with a constant correlation, Hadoop presents higher consumption for PageRank and Connected Components even when their runtimes are similar to TeraSort. This demonstrates that Hadoop is clearly less energy efficient when running iterative workloads that are not I/O-bound.

2.3.3. Power consumption and resource utilization

Analyzing the usage of system resources can provide a better understanding of both the processing frameworks and the workloads, being of great interest to correlate these system metrics with power consumption, as can be done with BDEv. We have selected three workloads for this analysis: TeraSort, K-Means and PageRank. Results for Connected Components have not been included as its power profile is very similar to PageRank. The power values shown in the next graphs only include the consumption of the entire package and the DRAM component (see Figure 2.2), as this CPU model (Intel Haswell) has neither an integrated graphic card nor does it provide the separate consumption of the processing cores. For simplicity purposes, the power consumption of each node is calculated by summing the

values of its two CPUs.

Figures 2.5, 2.6 and 2.7 present the power consumption, CPU and disk utilization for TeraSort, K-Means and PageRank, respectively. Regarding TeraSort, its I/O-bound nature can be clearly identified by analyzing the disk utilization graphs. However, the CPU graphs show isolated peak values for all the frameworks, especially at the end of the shuffle phase (around the second 200 for Hadoop and Spark, and 350 for Flink). As expected, those CPU peaks are correlated with the peak values in the power consumption graphs. It is interesting to note that Hadoop is not only the fastest framework for TeraSort, but also the one that shows a more stable power consumption, aside from two short peaks at the beginning of the execution and during the shuffle phase (see Figure 2.5a). Although the runtimes of Flink and Spark are very similar, they present quite different power, CPU and disk profiles. In fact, Flink shows higher CPU usages (see Figure 2.5f), averaging 70% CPU usage during the shuffle phase, while Hadoop and Spark never go above 30% aside from very short peaks. Hadoop presents the lowest CPU utilization, especially during the reduce phase (just after the shuffle phase), which explains its great energy savings mentioned before in the analysis of Figure 2.4b. Finally, it is easy to observe that Hadoop and Spark present a clear disk bottleneck during approximately the first half of the execution (see Figures 2.5g and 2.5h), while Flink also shows this bottleneck but especially during the computation of the reduce phase (see Figure 2.5i). This fact proves that the underlying data flow implemented by Flink to process the input dataset is quite different to the one used by Hadoop and Spark. Note that, as mentioned in Section 2.1.1, Hadoop and Spark are based on batch processing engines, while Flink relies on a stream processing architecture that treats batch workloads as a special case of streaming computations.

A similar analysis can be conducted for K-Means and PageRank (Figures 2.6 and 2.7). Their iterative nature can be seen in the cyclical behavior of both power consumption and CPU utilization. This is especially clear for Hadoop and Flink in K-Means (Figures 2.6d and 2.6f), and for Hadoop and Spark in PageRank (Figures 2.7d and 2.7e). Regarding the power consumption of K-Means, Spark shows considerably lower power values (below 90 watts) than Flink (see Figures 2.6b and 2.6c), which correlates with the aforementioned 10x improvement pointed out by the ED^2P ratio. The analysis of the system resource utilization illustrates that the

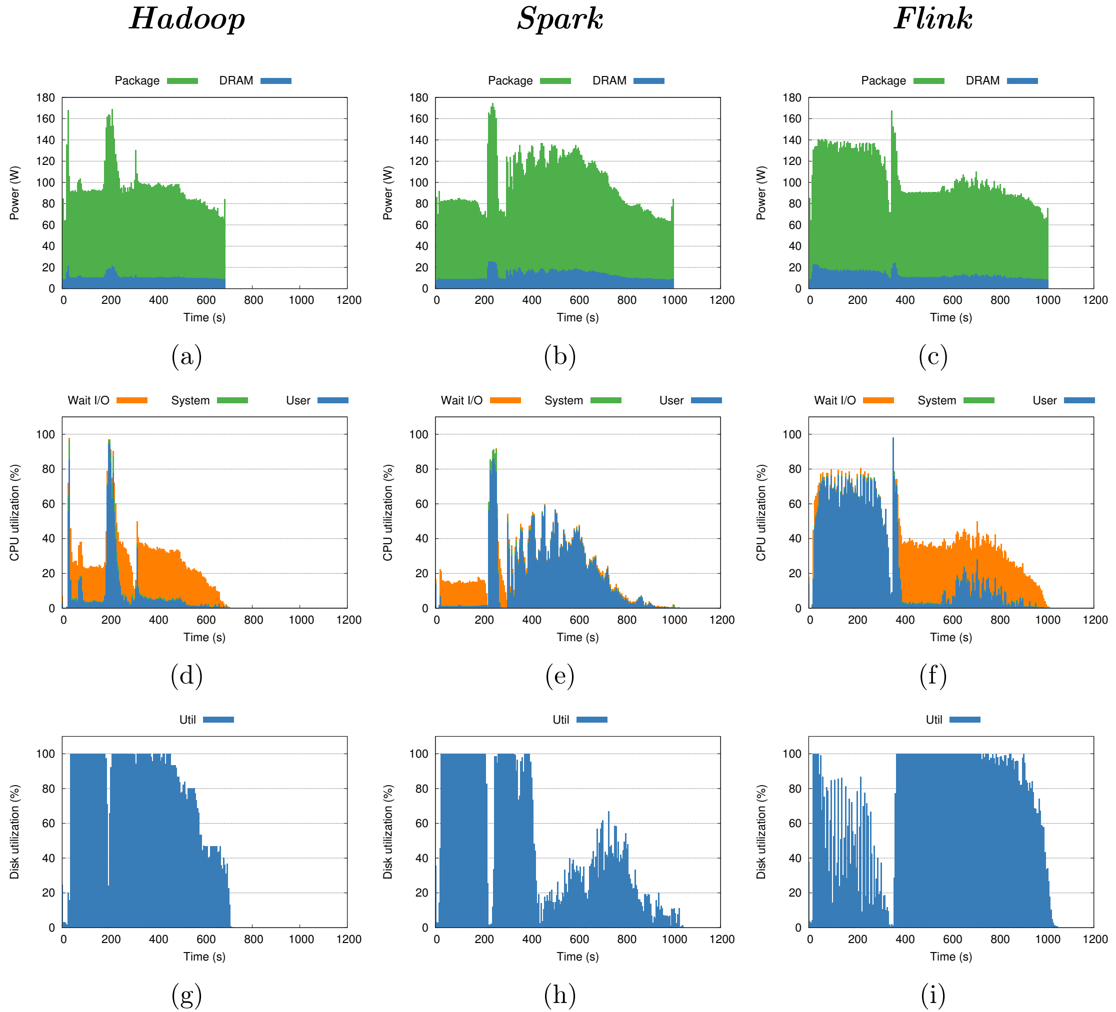


Figure 2.5: Average power consumption and resource utilization per node for TeraSort

disk access pattern for K-Means presents certain similarities across all the frameworks, showing noticeable disk I/O at the beginning and end of the execution, which mainly corresponds with reading/writing the input/output data (see Figures 2.6g, 2.6h and 2.6i). However, almost no disk activity is observed during the data processing, discarding any disk bottleneck in K-Means. This is mostly due to the strong iterative nature of this algorithm and the caching of intermediate results. Spark and Flink perform this caching by using in-memory data structures, while Hadoop takes advantage of the underlying buffer cache provided by the operating system that stores the data before writing to disk. In this case, it seems that

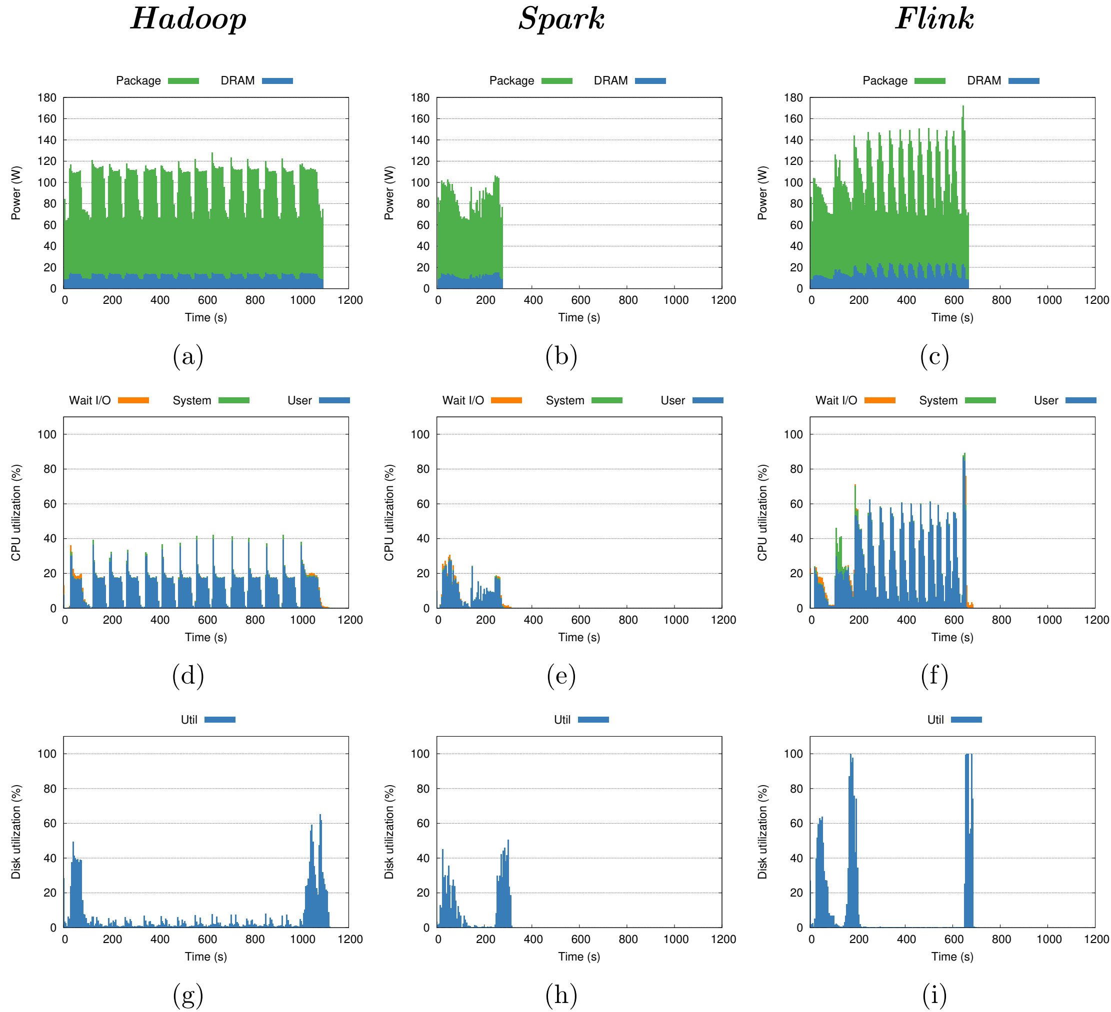


Figure 2.6: Average power consumption and resource utilization per node for K-Means

the amount of data being processed can be cached entirely in memory. Regarding PageRank (Figure 2.7), Hadoop and Spark present high to medium disk utilization, respectively, as shown in Figures 2.7g and 2.7h. This is especially relevant when compared with Flink, which shows nearly zero disk I/O except an initial short peak to read the input dataset (see Figure 2.7i). This means that the PageRank implementations of Hadoop and Spark are generating a higher amount of intermediate data than the Flink counterpart, thus causing higher disk I/O and runtime. Nevertheless, it can be concluded that no disk bottleneck occurs for Spark and Flink. Finally, Hadoop is not only the worst performer when running PageRank, but also

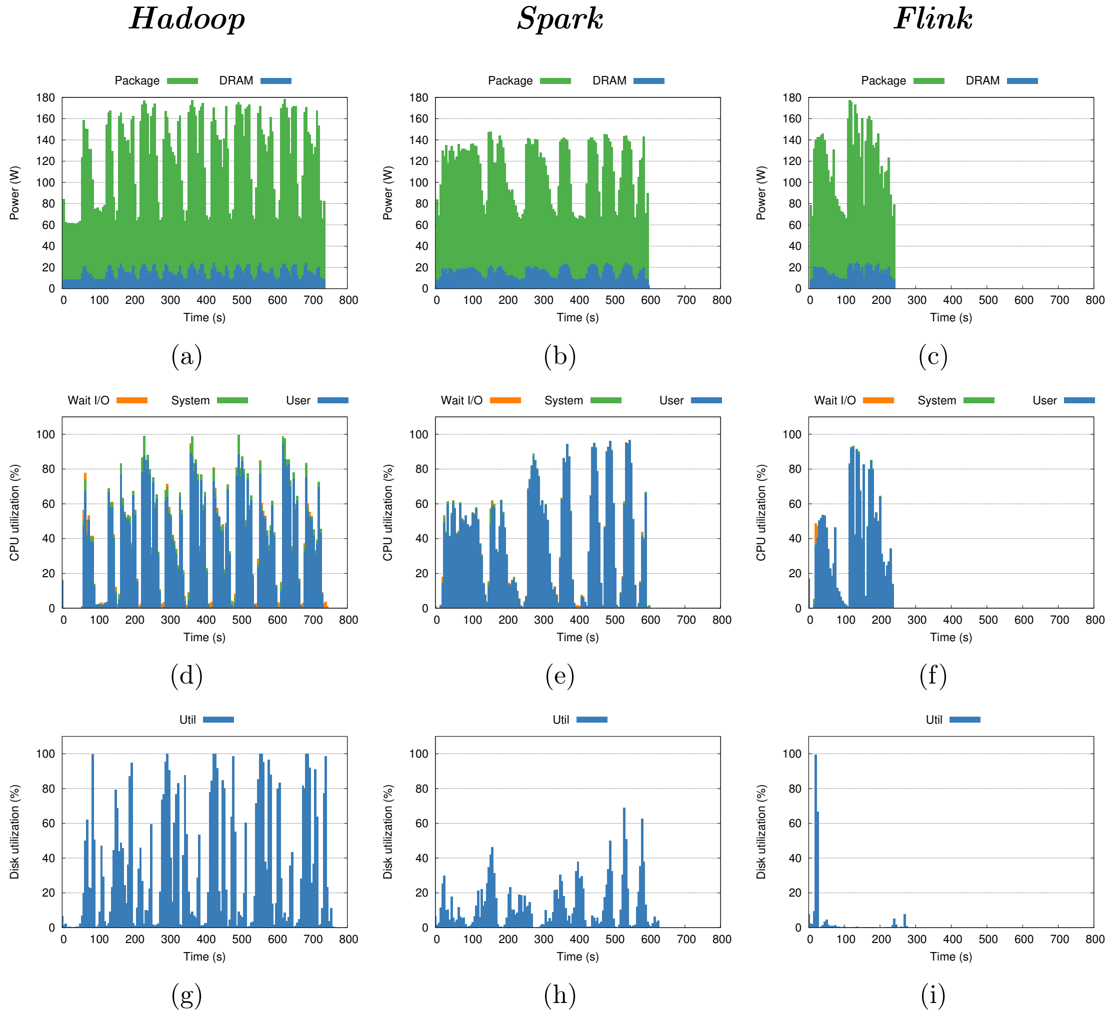


Figure 2.7: Average power consumption and resource utilization per node for PageRank

presents the highest power consumption, CPU and disk utilization by far. This confirms Hadoop as the least energy efficient framework when executing iterative workloads, as mentioned before when analyzing the energy results of Figure 2.4b.

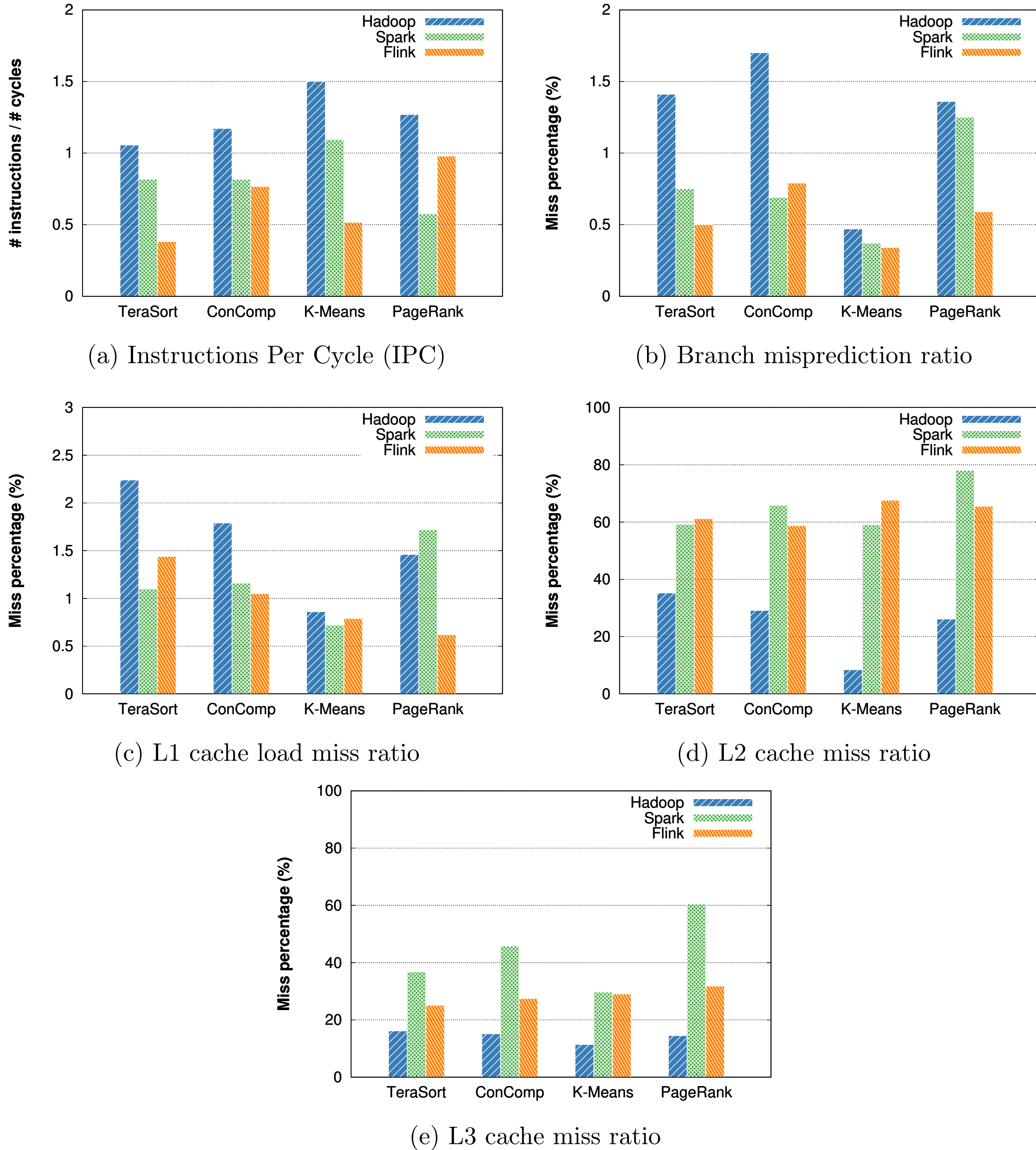
2.3.4. Microarchitecture-level metrics

The microarchitectural characterization has been carried out by measuring 10 hardware performance counters to calculate 5 different microarchitecture-level metrics: Instructions Per Cycle (IPC), branch misprediction ratio and L1/L2/L3 miss

ratios. The L1 total miss ratio is not supported by this CPU, and thus the L1 miss ratio only refers to load instructions. Figure 2.8 displays the results obtained for these metrics, calculated over the sum of the values recorded in all nodes of the cluster.

Regarding IPC (see Figure 2.8a), Hadoop outperforms both Flink and Spark for all the workloads, while Spark obtains higher values than Flink except for PageRank. The higher IPC obtained by Hadoop does not mean that it is actually performing better, as it only translates into more work being done per clock cycle. It is important to note that the efficiency indicated by the IPC metric may not be directly related with the execution time, as the high-level source code used to implement the frameworks (Java or Scala depending on the framework) and the workload itself are quite different (i.e. the machine code executed in the end by the CPU can vary completely). For the rest of the metrics, lower values generally translate into better performance. The branch misprediction ratios displayed in Figure 2.8b show that Hadoop presents higher values than Spark and Flink. In this case, a higher value can directly affect performance by decreasing the throughput of the CPU pipeline, as much more useless instructions are first being issued to the execution pipeline and then discarded when the branch prediction turns out to be wrong. Flink and Spark do not present any prevalence one over the other, aside from the PageRank implementation of Flink, that shows a noticeable lower value than Spark (highly correlated with the execution times shown in Figure 2.4a). Although a higher branch misprediction ratio may initially be related to a higher IPC value due to the additional instructions that are executed, the IPC shown in the graph is actually calculated using the INS_RETIRED event. This event only counts those instructions that end up being needed by the program flow. So, all those other instructions executed by the CPU in a speculative way are not taken into account, such as the ones that are issued when predicting a branch.

The miss ratios for the three cache levels available in this system (L1, L2 and L3) are shown in Figures 2.8c, 2.8d and 2.8e, respectively. For the first level, there is no clear difference between the frameworks, as this cache turns out to be highly efficient obtaining very low ratios overall (below 2.5%). As data is generally read in a sequential way when processed by these frameworks, most of these accesses hit the L1 cache. For the L2 cache, Hadoop presents lower miss ratios than Spark and



(a) Instructions Per Cycle (IPC)

(b) Branch misprediction ratio

(c) L1 cache load miss ratio

(d) L2 cache miss ratio

(e) L3 cache miss ratio

Figure 2.8: Microarchitecture-level metrics results

Flink, with a wide difference in K-Means (86% and 88% lower, respectively). For Spark and Flink there is no clear prevalent solution, as results are rather similar and depend on the workload. The higher L2 miss ratios of Spark and Flink means that, although they have shorter execution times for all the workloads except for TeraSort, their memory accesses seem to present less data locality. The same behavior can be

observed for the L3 cache, as Hadoop shows the lowest values. Furthermore, both Hadoop and Flink have rather constant L3 miss ratios across workloads, with values under 20% and 30%, respectively. Meanwhile, Spark shows higher and more variable ratios, ranging from 30% to 60%. PageRank presents the worst scenario with a considerably higher ratio than Hadoop and Flink. Note that this may be related to the PageRank execution time obtained by Spark, which is significantly worse than Flink as shown in Figure 2.4a. The low variability of the L3 miss ratios of Hadoop and Flink can be explained by the larger size of this last cache level compared with the other ones. This fact mitigates any difference that the implementation of the workload may present for each framework.

The analysis performed in this section has shown the utility of microarchitecture-level metrics to evaluate the execution of Big Data workloads. Other scenarios where the use of these metrics can be useful are: (1) comparing the behavior of different implementations of the same algorithm using a particular Big Data framework; (2) analyzing the impact of different configuration parameters of a framework; and (3) improving a specific microarchitectural metric when optimizing applications and frameworks.

2.4. Conclusions

Current Big Data applications are becoming increasingly larger and even more resource consuming. However, the performance metrics generally used to characterize Big Data processing frameworks, such as total execution time or average CPU utilization, are often simplistic. Although these metrics can be useful to provide a broad picture, the large scale of Big Data applications is demanding the consideration of more insightful metrics, like energy consumption or low-level microarchitecture events, to adequately assess the performance of both the frameworks and workloads. However, only some recent state-of-the-art evaluations focus on energy efficiency, while very little work takes into account microarchitecture-level aspects. Furthermore, none of them provides reliable, portable and publicly available tools to carry out the evaluations on different systems.

This chapter has first presented an overview of the state of the art regarding

the evaluation of distributed data processing frameworks for Big Data analytics, highlighting the need of more advanced evaluation tools to enable the assessment of resource utilization, energy efficiency and microarchitecture-level characterization. Next, we have presented the design of BDEv, a powerful benchmarking tool that allows the utilization of such metrics in multiple use cases, made by supporting dstat-based records, Intel RAPL measurements and Oprofile-based monitoring. As these utilities are available in a wide range of systems, the portability of the tool is ensured. Finally, BDEv has been used to evaluate representative data processing frameworks (Hadoop, Spark and Flink) in terms of performance, resource utilization, energy efficiency and CPU events. The analysis of the results provided by BDEv demonstrate its usefulness to fully understand the behavior of these frameworks when processing large datasets.

Chapter 3

Flame-MR: efficient event-driven MapReduce data processing

Most organizations that analyze their data with the MapReduce paradigm use the popular Hadoop framework. As mentioned in Section 2.1.1, the performance issues of its design have caused the development of several Hadoop modifications (e.g. RDMA-Hadoop), which attempt to improve performance by changing some of its underlying subsystems (e.g. network communications). However, these solutions are not always capable to cope with all its performance bottlenecks, as they only focus on certain parts of the Hadoop architecture. Therefore, the remaining parts can inherit existing inefficiencies such as excessive object creations or redundant memory data copies. The portability of the framework can also be affected if the optimizations are system-specific, like the use of native libraries or hardware accelerators. Moreover, Section 2.3 proved that up-to-date frameworks like Spark or Flink can achieve good performance improvements, although they do not keep compatibility with Hadoop applications. Adapting the source code of these applications (if possible) can be an overwhelming task, so it may not compensate the actual performance benefits obtained as a result.

For these reasons, there is a need for new frameworks in order to accelerate existing Hadoop applications while preserving API compatibility. This chapter introduces Flame-MR [127], a new event-driven MapReduce framework that redesigns completely the underlying Hadoop architecture. It enables efficient MapReduce data

processing by avoiding redundant memory copies and pipelining data movements, without requiring to modify the source code of existing applications.

Furthermore, Java applications often suffer from garbage collection overheads when the size of the heap managed by the Java Virtual Machine (JVM) reaches the available memory size [56]. This turns memory management into a crucial performance factor to be considered when developing Big Data frameworks. One of the main goals of Flame-MR is to maximize its in-memory data processing capabilities. So, it must manage memory resources in an efficient way to minimize garbage collection overheads. This chapter addresses this issue by presenting some proposals in order to reduce memory allocations, while also caching intermediate results to avoid unnecessary network and disk operations. The impact of these proposals on the performance of Flame-MR is evaluated through experimental analysis.

This chapter is organized as follows. Section 3.1 describes the main characteristics of the MapReduce programming model and its de facto standard implementation, Hadoop. Section 3.2 analyzes other existing works that attempt to optimize the performance of Hadoop by utilizing different approaches. The design of Flame-MR is next described in Section 3.3, giving some details about the most important optimizations implemented. Section 3.4 analyzes the memory usage of Flame-MR and describes some further optimizations. Finally, the conclusions extracted are commented in Section 3.5.

3.1. Background

The MapReduce programming model was originally proposed by Google in [29]. This model allows developing large-scale Big Data workloads by keeping low-level implementation details such as task parallelization and data communication hidden to the programmer. The only thing that has to be defined are the data processing functions, map and reduce, that operate the input data represented in form of key-value pairs. The map function processes each input pair independently to extract the relevant attributes and the reduce function operates them to get a final result.

As mentioned in Section 2.1.1, the most popular implementation of MapReduce is Hadoop [8], an open-source Java-based framework frequently used for storing

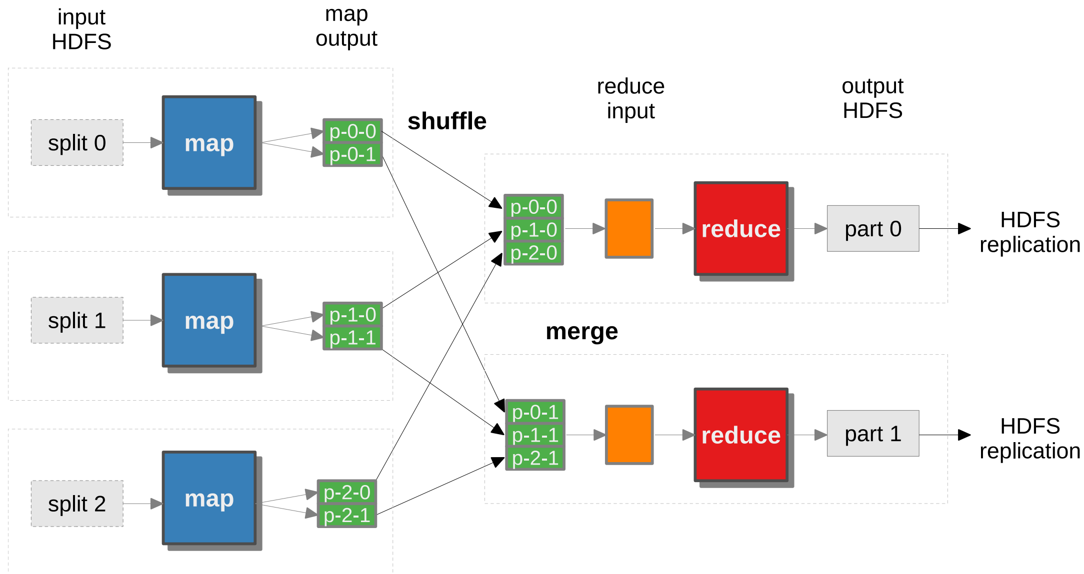


Figure 3.1: Hadoop data flow with multiple map and reduce tasks

and processing large datasets. It mainly consists of two components, the MapReduce data engine and the Hadoop Distributed File System (HDFS) [108], which distributes the storage of large amounts of data over the nodes of a cluster. Hadoop workloads commonly use the MapReduce model to process textual data stored in HDFS, following several steps: input, map, shuffle, merge, reduce and output. These steps are depicted in Figure 3.1. As can be seen, the input dataset stored in HDFS is divided in many splits that are read by map operations to extract the relevant key-value pairs. These pairs are partitioned, sorted by key and sent through the network to the nodes where they will be merged to form the reduce input. Each reduce operation reads the pairs contained in its input partition, processing them to generate the output result that is written to HDFS.

Hadoop can adapt its behavior to the particular needs of each application, as it provides a wide set of configuration options to do so. This includes the setting of some software components defined via Java interfaces, modifying their implementation according to the computation that the user needs to perform. For example, the user can configure a different input and output formatter class if the data is not in textual format. Similarly, users can use primitive data types included in Hadoop or define their own ones by developing a custom implementation of the Writable interface. This interface establishes the methods that the custom data

types need to implement, which are mandatory to serialize and compare the data objects. Moreover, other application types apart from MapReduce can be deployed in a Hadoop cluster by using Yet Another Resource Negotiator (YARN) [121], which was introduced in Hadoop 2 to manage the computational resources of the nodes.

Nowadays, many applications and libraries use Hadoop to carry out MapReduce workloads. However, Hadoop presents some performance bottlenecks that hinder its utilization for large-scale analytics due to poor resource utilization and inefficient data parallelism. This situation has caused the appearance of several alternative frameworks like Spark and Flink, which can be used to execute Big Data workloads with a more flexible API and increased performance. However, rewriting existing MapReduce applications to the new APIs generally requires a significant programming effort. Furthermore, the source code is not always publicly available, which precludes the users from rewriting it. This Thesis overcomes these issues by proposing Flame-MR, a new MapReduce framework that redesigns completely the Hadoop architecture in order to improve its performance and scalability while keeping compatibility with Hadoop APIs. Furthermore, its Java-based implementation ensures its portability.

3.2. Related work

The broad adoption of the Hadoop project has caused the appearance of several MapReduce frameworks that attempt to improve its performance. Most of them modify some of its subsystems, like network communications or disk I/O, to adapt them to specific environments. That is the case of Mellanox Unstructured Data Accelerator (UDA) [86] and RDMA-Hadoop [57], which adapt Hadoop to HPC resources, such as RDMA interconnects like InfiniBand. On the one hand, Mellanox UDA is a plugin written in C++ which combines an RDMA-based communication protocol along with an efficient merge-sort algorithm based on the network levitated merge [131]. On the other hand, RDMA-Hadoop redesigns the network communications to take full advantage of RDMA interconnects, while performing data prefetching and caching mechanisms [132]. RDMA-Hadoop incorporates these modifications in a Hadoop distribution which is available separately. Both Mellanox UDA and RDMA-Hadoop keep compatibility with the user interfaces. However, they

only modify certain Hadoop subsystems, which can lead to limited performance improvements compared to an overall redesign of the Hadoop underlying architecture.

Another modification of Hadoop is NativeTask [141], which rewrites some of its parts using C++, like task delegation and memory management. Furthermore, it takes into account the cache hierarchy to redesign the merge-sort algorithm [140]. However, the optimizations performed by NativeTask are highly dependent on the underlying system, which hinders its portability. This is also the case of Main Memory MapReduce (M3R) [107], which uses the X10 programming language [20] to implement an iterative MapReduce framework that keeps compatibility with Hadoop APIs. Another important drawback of M3R is that the workload has to fit in memory, preventing its use for real-world Big Data scenarios. iMapReduce [147], another iterative Hadoop modification, not only avoids to write intermediate results to HDFS, but also minimizes task scheduling and synchronization overheads. Nevertheless, the applicability of these optimizations is limited to applications with single-job iterations.

The performance bottlenecks of Hadoop have caused the emergence of new frameworks that fully replace the Hadoop implementation. One of these frameworks is DataMPI [78], which makes use of the MPI standard to leverage the high-performance interconnects that are usually available in HPC systems. MapReduce Implementation Adapted for HPC Environments (MARIANE) [36] is designed to take advantage of the General Parallel Filesystem (GPFS), which is also commonly found in HPC systems. Other solutions, like Spark [145] and Flink [7], optimize the memory usage by using collections of elements as an alternative to key-value pairs. Both expand the set of operations available to the end user, rather than providing only map and reduce functions. The main problem with this kind of proposals is that they do not provide full compatibility with Hadoop APIs, so the code of the applications must be adapted or even rewritten from scratch.

Memory efficiency has been the subject of study of many works. Some of them [83, 113] enhanced the scalability of MPI libraries in terms of memory usage. Although these works are not focused on Big Data processing, some of the challenges they face are similar to our case (e.g. keeping memory scalability as the number of processes grows). Regarding Big Data frameworks, there exist some previous works that have investigated their memory efficiency. In [32], the memory usage

of Hadoop and Hive is characterized by analyzing the memory footprint, memory bandwidth and cache misses of different workloads. The authors of [95] studied the use of large memory pages when executing Big Data applications in non-uniform memory access systems, concluding that large pages do not show significant performance improvements when the dataset is sufficiently large. The scalability of Spark has been evaluated in [142], studying whether it is better to increase the number of nodes in the cluster or improve the hardware characteristics (CPU, memory and disk) of the nodes. Although each configuration performed differently depending on the workload type, the one with faster nodes showed a better performance per watt ratio.

The development of new techniques to improve memory management in Big Data frameworks has been addressed in different ways. In [139], the authors presented a novel two-level storage system with an upper-level in-memory file system that leverages high I/O throughput and data locality, while a lower-level in-disk parallel file system provides consistency and larger capacity. Moreover, works like [25] use new storage technologies, such as SSD disks, to alleviate bandwidth and memory requirements. Other proposals target the memory management performed by object-oriented languages (e.g. Scala, Java) in order to adapt it to the characteristics of Big Data workloads. That is the case of Yak [92], which proposes a hybrid garbage collection approach that distinguishes between objects belonging to the control space and the data space. Yak adapts the memory management of each object type to its lifespan characteristics, being able to alleviate the overhead of garbage collection operations in Big Data frameworks like Hadoop. Another work, Deca [77], analyzes the data objects of Spark applications to estimate their expected lifespan, allocating and releasing memory accordingly to minimize garbage collection overheads. Although it would be interesting to compare these alternatives with Flame-MR, these projects are not publicly available.

Finally, another modification of Hadoop, called SHadoop [54], also improves the job and task execution mechanisms, although the improvement is mostly significant in jobs with short runtimes.

3.3. Flame-MR design

This section presents the overall design of Flame-MR. First, the main characteristics of its internal architecture are discussed in Section 3.3.1. Second, Section 3.3.2 describes in more detail the different phases of the MapReduce data processing pipeline in this architecture.

3.3.1. Flame-MR architecture

Flame-MR is a distributed processing framework implemented in "pure" Java code (i.e. 100% Java) for executing standard MapReduce algorithms. Being fully integrated with the Hadoop ecosystem, Flame-MR runs on YARN, which is its resource management layer, and uses HDFS for data storage. Its design is oriented to optimize the performance of the overall MapReduce data processing, improving the utilization of the system resources (CPU, memory, disk and network) and the overlapping of the data flow. Moreover, the architecture of Flame-MR has a strong flexibility due to the use of the same software interfaces to manage in-memory data, network communications and HDFS I/O. Flame-MR acts as a plugin that is fully compatible with Hadoop APIs, so existing MapReduce applications do not have to be rewritten.

The Flame-MR workflow is composed of the classic MapReduce phases: input, map, sort, copy (or shuffle), merge, reduce and output. The input phase reads the input dataset from HDFS and the map phase extracts the valuable information by applying the user-defined map function to each input pair. Once the map output is generated, the sort phase ensures the correct ordering of the output pairs, which are sent through the network during the copy step. The merge phase generates the reduce input by merging all the incoming map output pairs. Next, the reducer applies the user-defined reduce function to each set of key-value pairs, computing the final output which is written to HDFS in the output step. In Hadoop, one or more phases are performed for a certain part of the input dataset by independent Java processes called tasks (e.g. a map task performs the input, map and sort phases). However, Flame-MR arranges the phases into MapReduce operations, which are logical processing units performed by a Java thread. Unlike Hadoop, these operations

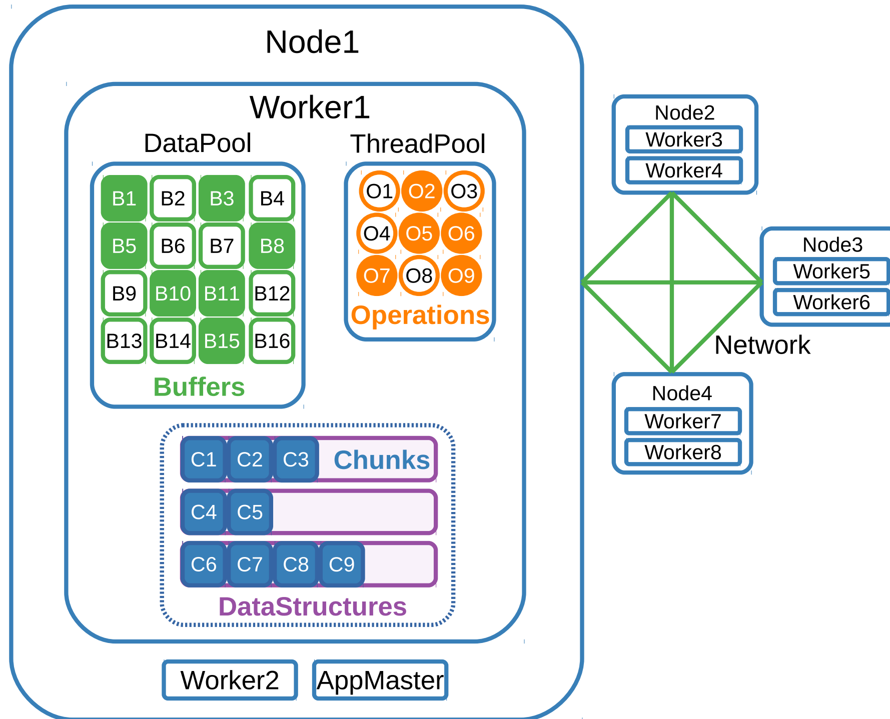


Figure 3.2: High-level architectural overview of Flame-MR
O: Operations B: Buffers C: Chunks

are executed within the same Java process (from now on, Worker). For a given input data, each operation performs some of the explained phases depending on its operation type (map, merge or reduce), as will be described in Section 3.3.2.

Figure 3.2 presents a high-level overview of the Flame-MR architecture, which is based on a traditional master-slave model. This model has been adapted to YARN, in which the master and the slaves are executed inside YARN containers. At the application launching, the master container (AppMaster in the figure) allocates one or more Workers per computing node in the cluster as configured by the user. In the same way, the user configures the CPU cores and memory allocated to each YARN container, which in turn determines the resources available for the Workers. The configuration of the Workers (i.e. number of Workers per node and resources available for the Worker) provides higher flexibility than the Hadoop model, in which each map/reduce task is allocated in a separate container depending on the amount of memory available in each node. Furthermore, each Worker in Flame-MR allocates memory and CPU resources by means of a DataPool and a ThreadPool, respectively.

The DataPool structure manages the amount of memory available in the Worker, allocating memory buffers for the different MapReduce operations. Likewise, these operations allocate CPU cores by means of the ThreadPool scheduler.

The input and output data is read and written, respectively, by using DataStructures. Basically, these structures are data queues that contain a certain number of data chunks to be processed. Moreover, the Workers use the network to transfer the map output data and synchronize their computation at certain moments of the process (e.g. before the reduce phase). More details about the architectural design of Flame-MR are provided next.

Optimized memory usage

Flame-MR manages in-memory data using chunks, which are treated as logical data units in the same way as HDFS uses blocks. The chunks are written by an operation in a MapReduce phase and read by another operation in the next phase. Internally, each chunk has a number of memory buffers that contain the physical data in memory or abstract a file path in disk. In order to manage the buffers that fit in memory, each Worker defines a DataPool for allocating them. This pool is shared among all the MapReduce operations in a certain Worker, optimizing the amount of memory needed at any moment. When a configurable threshold size is exceeded, the DataPool begins to spill buffers to disk, allocating the freed space to new buffers, until the threshold condition is satisfied again. After that, if a spilled buffer is attempted to be read, the DataPool reallocates its original size and the data is read back from disk.

All buffers have a fixed size, which is configurable by the user (the default value is 1 MB). The configuration of the buffer size is useful in order to handle the available memory and balance the number of buffers managed by the DataPool during the MapReduce data processing. Moreover, the DataPool uses a single operation to spill a buffer to disk or read it back, so the buffer size is a relevant parameter in order to optimize the I/O throughput.

Thread-based processing model

In Hadoop, each map/reduce task is executed in a YARN container (i.e. in a JVM), which has a separate resource allocation. This can lead to poor resource utilization at some stages during the MapReduce workflow. For example, reducers which are waiting for the mappers to complete cannot share their resources with them and so the finalization of the mappers is delayed. Flame-MR differs from this model by treating each map/reduce task as an operation, which is carried out by a Java thread. Moreover, the overhead of thread creation/destruction is minimized by means of a ThreadPool which runs as many threads as the number of CPU cores available for the Worker. As in the case of the DataPool, the ThreadPool is shared by all the MapReduce operations in a Worker, optimizing the number of them being executed at any moment and maximizing the memory utilization and CPU resources.

The ThreadPool has a queue of operations waiting to be processed, which are ordered by using a priority system. The priority of each operation is determined by the MapReduce phase to which it belongs. For example, map operations have a higher priority than merge operations, in order to maximize the amount of map outputs that can be processed by the merge operations. Further details about the operations performed by Flame-MR are given in Section 3.3.2.

Event-driven architecture

The relationship between data chunks and the operations that process them is made through an event-driven architecture. This architecture is composed of several DataStructures that contain the data chunks waiting to be processed. Depending on the specific MapReduce phase, the pending operations are queued to the ThreadPool to process all the input data chunks existing in a DataStructure (e.g. at the beginning of the reduce phase) or to process the chunks as they arrive to the DataStructure (e.g. in the input of the merge phase). In the latter case, when the DataStructure receives new data chunks it generates an event to indicate the ThreadPool that there are data waiting to be processed.

There are three main types of DataStructures: in-memory, network and HDFS.

In-memory DataStructures (i.e. the input for merge and reduce operations) contain several data chunks which are waiting to be processed. They behave as a queue between MapReduce operations (e.g. between merge and reduce operations in the case of the reduce input DataStructure, which will be further depicted in Figure 3.4). Network DataStructures abstract network communications (i.e. the map output). Data chunks added to these DataStructures will be sent through the network to the corresponding Worker, which is determined by the partition assigned to the chunk. Finally, HDFS DataStructures abstract I/O movements to HDFS, reading the input at the beginning of the MapReduce workflow (i.e. the map input) and writing the output at the end (i.e. the reduce output). These three kinds of DataStructures implement the same interface, which provides a flexible software architecture.

The design of the event-driven architecture in Flame-MR keeps some similarities with the Staged Event-Driven Architecture (SEDA) [135]. SEDA proposes an architecture for well-conditioned, scalable Internet services consisting of several stages connected by queues, in which each stage represents a step of the whole process. Flame-MR also carries out a series of stages by means of the MapReduce operations, which are connected by DataStructures that work in a similar way as the queues in SEDA. However, Flame-MR uses the same ThreadPool for all the operations instead of using one ThreadPool per stage as in the case of SEDA. By doing this, Flame-MR optimizes the number of operations that are being executed at any moment and prioritizes them in order to maximize the amount of data processed.

Copy-avoidance mechanism

One of the main performance bottlenecks of Hadoop is the high number of redundant memory copies. Many of them are performed when reading from HDFS and translating the input data stream to Writable objects (e.g. Text, IntWritable, LongWritable). In order to alleviate this situation, equivalent primitive Writable types in Flame-MR do not copy data fields from the input chunks to the corresponding memory objects, but keep references to these fields instead. Using the position where each field starts and its length, the information is not retrieved and translated unless it is needed, avoiding extra data copies. Furthermore, data fields do not have to be translated into the Writable objects when written to another data chunk. This is especially relevant in identity map/reduce functions, which write their input

key-value pairs to the output chunks without modifying them. In this kind of functions, which are very common in some workloads (e.g. Sort), the references kept by the Writable objects are used to copy each data field directly from the input to the output chunk, and thus involving a single memory copy.

Hadoop-like map and reduce functions

Existing Hadoop applications can be executed without any source code modification, as Flame-MR implements the Hadoop APIs. An example of the source code defined by the user is presented in Figure 3.3, which shows the map and reduce functions (Figures 3.3a and 3.3b, respectively) for the WordCount program. Note that there is no difference between the source code of both Flame-MR and Hadoop functions, so they are displayed only once. Likewise, the Hadoop driver of the MapReduce job does not need any change to be run with Flame-MR.

3.3.2. MapReduce operations

As mentioned before, the MapReduce data processing in Flame-MR is divided into three operation types: map, merge and reduce. These operations can be observed in Figure 3.4, which shows an overview of the whole data processing pipeline. More details about the computation that these operations perform and their specific optimizations are provided next.

Map operations

At the beginning of the job execution, the AppMaster (see Figure 3.2) divides the input dataset into independent splits, and then they are allocated to the different Workers. Next, each Worker creates one map operation per input split and queues them to the ThreadPool. Each map operation processes the input, map, sort and copy phases for its associated split. The basic stages of a map operation are depicted in the left part of Figure 3.4a. In the input step, the map operation (O1 in the figure) creates a new data chunk (C1) and uses the DataPool to allocate enough space for it. The chunk is filled with the data using the HDFS client libraries. Once the input

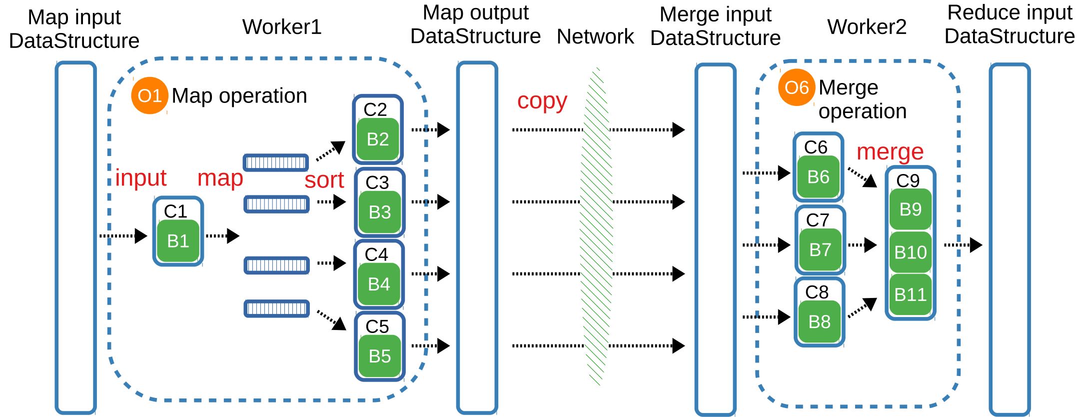
```
public static class WordCountMapper extends
Mapper<Object, Text, Text, IntWritable>
{
 private final static IntWritable one = new IntWritable(1);
 private Text word = new Text();

 public void map(Object key, Text value,
 Context context) throws IOException, InterruptedException
 {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens())
  {
   word.set(itr.nextToken());
   context.write(word, one);
  }
 }
}
```

(a) WordCount mapper

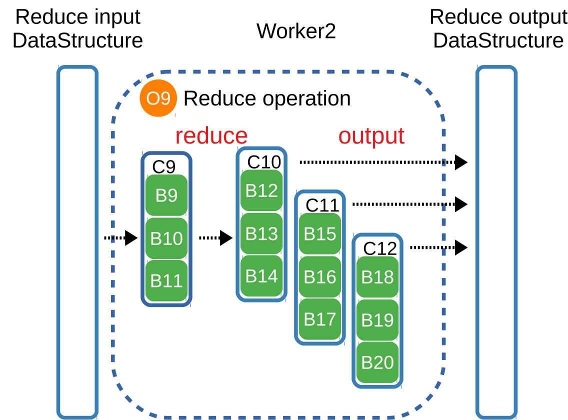
```
public static class WordCountReducer extends
Reducer<Text, IntWritable, Text, IntWritable>
{
 private IntWritable result = new IntWritable();

 public void reduce(Text key, Iterable<IntWritable> values,
 Context context) throws IOException, InterruptedException
 {
  int sum = 0;

  for (IntWritable val : values)
  {
   sum += val.get();
  }

  result.set(sum);
  context.write(key, result);
 }
}
```

(b) WordCount reducer

Figure 3.3: Code examples for WordCount map and reduce functions in Flame-MR and Hadoop



(a) MapReduce data flow of map and merge operations



(b) MapReduce data flow of reduce operations

Figure 3.4: Overview of the MapReduce workflow in Flame-MR
O: Operations B: Buffers C: Chunks

chunk resides in an in-memory buffer (B1), its key-value pairs are read one by one and passed to the user-defined map function. As the map function generates the output key-value pairs, these are kept in memory and grouped by their partition number. This number is calculated based on the key of the pair, which determines the Worker that will merge and reduce it. Once the size of the output pairs exceeds a certain threshold, a partition group is sorted in memory and written to a new data chunk. During the copy stage, this data chunk is sent through the network to the corresponding Worker. Further characteristics of map operations are explained next.

- **Two-level partitioning**

  In Hadoop, each map output pair belongs to a partition that corresponds with the reducer that will process it. In Flame-MR, each Worker can run more than one reduce operation, and so each Worker has a set of partition numbers that belongs to it. Hence, there are two levels of partitioning: the first one defines the Worker that will reduce the partition and the second one defines the reduce operation within the Worker. This two-level partitioning is necessary in order to parallelize the reduce phase, while enabling the Worker to share the computational resources allocated to it among the different partitions.

- **In-memory object sort**

  Unlike Hadoop, map output pairs are not written to a temporary chunk while they are not sorted. As mentioned before, the Writable objects produced by the user-defined map function are kept in memory. When the size of the objects exceeds a certain threshold, they are sorted and written to the final data chunk that is sent through the network. This threshold is calculated as a percentage of the DataPool size divided by the number of threads in the ThreadPool. This value can be tuned by the user via a configurable parameter to adapt it to the characteristics of the underlying system, although a default value is provided. The in-memory object sort allows Flame-MR to avoid many memory copies during the sort phase. Furthermore, the copy-avoidance mechanism explained in Section 3.3.1 enables to store the output of identity map operations as objects that reference the input data, copying directly the input data to the output chunk when sorting the results.

- **Pipelined map, sort and copy phases**

  As explained in the previous paragraph, mappers sort and send their output pairs through the network once they exceed a certain threshold size. This behavior acts as a pipeline that alternates the computation of the map, sort and copy phases. The design of Hadoop takes a different approach, as the reducers are responsible for retrieving the output. The mappers sort and store their output locally until it is requested, by means of a network service called ShuffleHandler. This service is shared by the mappers of a node and

so it is placed in a separate Java process. Moreover, if a mapper runs out of memory, it will spill some of the output pairs to disk, reading them back when the reducer queries the map output. As Flame-MR map operations send directly their output chunks from the memory space of the Worker, the locality of in-memory data is higher compared with Hadoop. Flame-MR also avoids waiting for the reducers to retrieve the map output, which can delay the communications and cause unnecessary spill operations. However, this feature may have some impact on the fault tolerance of Flame-MR. As the mapper does not retain the map output pairs, they cannot be retrieved back if a node fails like in the case of Hadoop. Nevertheless, our first prototype focuses primarily on performance aspects of MapReduce applications. Further work is planned to study how to improve the fault tolerance of Flame-MR without harming performance.

The chunks are sent through the network by sending each data buffer to the destination. Flame-MR can be configured to send each data buffer in several packets, using a fixed packet size. The use of the packet size is useful to adapt the communications to the characteristics of the underlying network (e.g. maximum transmission unit), while the buffer size can be optimized for spilling to disk.

Merge operations

Merge operations are responsible for processing the map output chunks received through the network, merging them to form the reduce input. The data flow of merge operations can be seen in the right part of Figure 3.4a. When a Worker receives a map output chunk, it is stored in a DataStructure and a new merge operation (O6 in the figure) is queued to the ThreadPool. When this operation is executed, it first checks the DataStructure for unprocessed partitions, selecting one of them and taking all its incoming chunks. Then, the operation merges these chunks until getting a single one, which is added to the reduce input DataStructure. Moreover, this chunk can be merged again as new chunks arrive at the Worker. Next, we provide some further details about the optimizations implemented in merge operations.

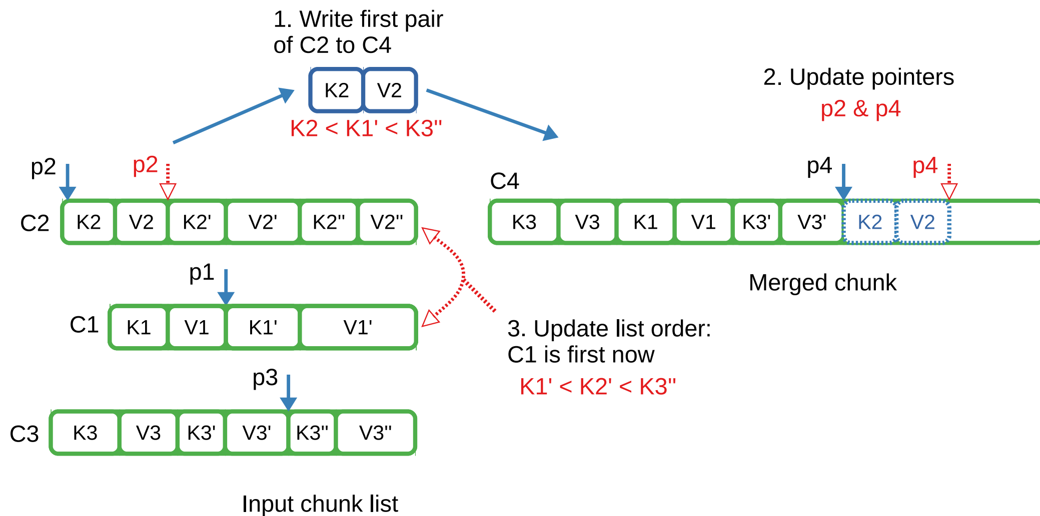


Figure 3.5: k-way merge (k=4)
C: Chunks K: Keys V: Values p: pointers

- **k-way merger**

  The merge algorithm used in Hadoop is a 2-way algorithm that merges two pairs at a time. Flame-MR uses a different approach by performing a k-way merge algorithm that merges k chunks at a time. The input chunks are organized in a list and sorted by the first key that appears in each chunk. Next, the first pair of the first chunk is copied to the merged chunk, advancing the position of the read chunk and reordering the chunk list according to it. The pairs are read until there is none left. An example of this algorithm is depicted in Figure 3.5, which shows the input chunks C1, C2 and C3 being merged to chunk C4. Each chunk contains a list of key-value pairs <Kx,Vx>. It can be seen that the first chunk of the list is C2, which is currently pointing to the key-value pair <K2,V2>. Hence, this pair is copied to chunk C4 in step 1. Next, the current positions of C2 and C4 are moved forward by updating their pointers (p2 and p4, respectively) in step 2. Finally, step 3 updates the list order by placing C1 in the first place as <K1',V1'> becomes the lowest key in the list.

  As the k-way merge algorithm processes multiple data chunks at once, it minimizes the number of merge operations needed to compute the reduce input, thus avoiding many comparisons and memory copies. Furthermore, Flame-MR

calculates k based on a percentage of the available memory, which enhances memory efficiency and avoids spilling data to disk.

- **Binary comparison**

  In Hadoop, many data types define the BinaryComparable interface. This enables to compare the binary representations of the objects in the merge phase using the memory buffers that contain their data, without translating the data fields to the Writable objects. Flame-MR also uses binary comparison, but it does not create the objects. Instead, it compares the fragments of the chunks corresponding to the objects. By doing so, it avoids creating and destroying many objects and the corresponding overhead of the JVM garbage collection.

Reduce operations

Reduce operations perform two phases: reduce and output. Their data flow is shown in Figure 3.4b. Once all map outputs have been merged, each Worker generates a reduce operation per chunk partition. Each reduce operation (O9 in the figure) reads its associated input data chunk and reduces the final output by applying the user-defined function to each set of key-value pairs. Finally, the chunks are stored in HDFS during the output phase. This phase has been optimized as explained below.

- **Pipelined output**

  Hadoop writes the reduce output pairs to HDFS as they are being generated by the reduce function. This causes a large number of writes to HDFS, with a significant overhead each of them. In order to avoid this, reduce operations in Flame-MR store their output pairs to a data chunk, which is written to HDFS once it reaches a certain threshold size. This mechanism is similar to the way map output pairs are sent through the network. The use of the threshold size ensures that the reduce output does not exceed the available memory space in the DataPool, in which case the data buffers would have to be spilled to a local disk. Furthermore, it also acts as a write pipeline that allows to alternate the computation of the reduce function and the writing of the results to HDFS.

3.4. Memory management optimizations

Memory usage is a crucial factor for avoiding performance issues when using large memory spaces. This section focuses on the development of efficient memory management techniques to analyze their impact on overall performance. Using the original implementation of Flame-MR presented in Section 3.3 as a baseline, several optimizations have been developed and evaluated comparatively through experimental analysis. First, Section 3.4.1 analyzes the performance overhead caused by long pauses of the Garbage Collector (GC), explaining several optimizations to reduce them. Second, Section 3.4.2 analyzes different implementations of the in-memory buffers that contain the data to be processed. Finally, Section 3.4.3 focuses on iterative workloads, explaining how to avoid the writing of intermediate results to HDFS and the benefits of doing so.

The optimizations presented in each of these sections are evaluated by means of several performance experiments carried out in a multi-core cluster, Pluton. Table 3.1 contains the main hardware and software characteristics of this cluster. According to these characteristics, Flame-MR has been configured as shown in Table 3.2, which also contains the relevant configuration parameters for HDFS. Moreover, Flame-MR has been configured to use the IPoIB interface, which allows to employ this network through the IP protocol and obtain lower latencies and higher bandwidths than with Gigabit Ethernet. The version of Hadoop deployed to make use of YARN and HDFS components is 2.7.2. The experiments have been carried out using 9 nodes, corresponding with 1 master and 8 slaves.

The execution of the experiments has been automated by using BDEv, selecting Sort and PageRank among the workloads supported as they are representative examples of I/O-bound and iterative workloads, respectively (see Section 2.2.3). Sort has been used to order a 100 GB input text dataset, while the input dataset of PageRank has a size of approximately 20 GB (30 Mpages), with a maximum of 5 iterations per execution. The results provided in this section take into account a minimum of 10 executions for each experiment making sure to clear the OS buffer cache between each execution. Finally, the GC used is Parallel GC, which is enabled by default and provides the best throughput as the number of cores increases [46].

Table 3.1: Node characteristics of Pluton

Hardware configuration	
CPU model	2 × Intel Xeon E5-2660 (Sandy Bridge-EP)
CPU Speed (Turbo)	2.20 GHz (3 GHz)
#Cores	16
Cache (L1 / L2 / L3)	32 KB / 256 KB / 20 MB
Memory	64 GB DDR3 1600 MHz
Disk	1 TB HDD
Networks	InfiniBand FDR & GbE
Software configuration	
OS version	CentOS release 6.4
Kernel	2.6.32-573
Java	Oracle JDK 1.8.0_45

Table 3.2: Configuration of Flame-MR in Pluton

Flame-MR	
Workers per node	4
ThreadPool size	4
Worker heap size	10.5 GB
DataPool size	6.3 GB
DataBuffer size	1 MB
HDFS block size	128 MB
Replication factor	3

3.4.1. Garbage collection reduction

The JVM provides automatic memory management by allocating available memory on the heap to objects when they are created. The programmer cannot control when this memory is released, as Java objects cannot be explicitly destroyed. Instead, the JVM tracks the objects on the heap and their references from the Java code, removing an object and releasing its memory when it has no other objects referencing to it. This process is transparently performed by the GC included in the JVM [114]. Using large heap sizes, garbage collection can consume a considerable amount of computational resources, incurring significant performance penalties and even program stalls. In fact, it is one of the most common performance issues in Java applications [56]. For Big Data frameworks, this issue becomes even more important, as they execute workloads with very long execution times, creating a large number of objects and having high memory usage. For this reason, Flame-MR has been analyzed to identify means of reducing the amount of garbage collection performed, which has led to two main techniques, static memory allocation and buffered map output, explained next. These techniques have been incorporated into a modified version of Flame-MR, called Flame-MR-GCop.

Static memory allocation

In the original implementation of Flame-MR, MapReduce operations (e.g. map, merge) allocate memory space by requesting DataBuffers to a DataPool. Each of these requests causes a new object to be created, with its corresponding memory allocation on the heap. Once the DataBuffer is used, it is returned to the DataPool, which dereferences the DataBuffer in order to be garbage collected. This behavior involves a lot of DataBuffer object creations and collections, with their corresponding memory allocations and deallocations. Moreover, the default implementation of DataBuffers relies on primitive byte arrays (i.e. *byte*[]) to contain the data, which are initialized to a default value at the time of creation. In Flame-MR-GCop, the behavior of the DataPool has been completely redesigned, using static memory allocation in order to avoid the excessive creation of DataBuffers and to minimize GC overheads.

The basic principle of static memory allocation is that once the memory has

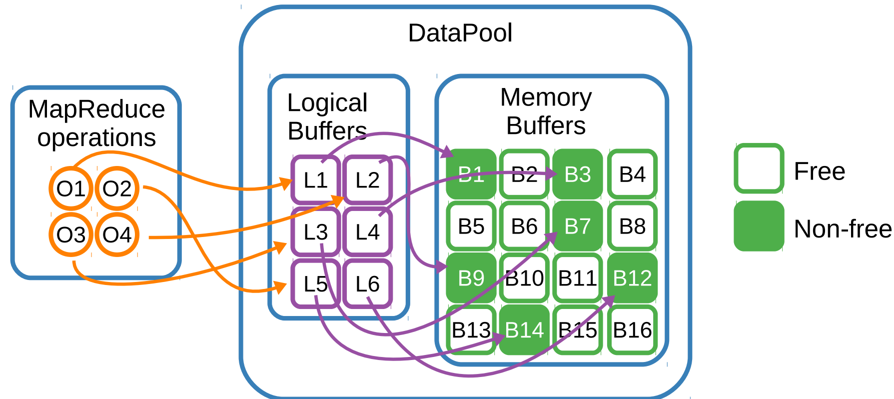


Figure 3.6: DataPool overview in Flame-MR-GCop

been allocated, it is reused by several operations, without neither releasing it nor dereferencing the objects. Figure 3.6 depicts the new design of the DataPool, where DataBuffers are managed by means of two different classes: LogicalBuffers and MemoryBuffers. On the one hand, LogicalBuffers are associated with the data they contain, which can be in memory or in disk. On the other hand, MemoryBuffers are associated with the memory space that is allocated to them, each one containing a byte array of a fixed size (1 MB by default). LogicalBuffers reference to Memory-Buffers, using them to store the data from the operations. When a LogicalBuffer is created by an operation, it requests a MemoryBuffer to the DataPool. If the DataPool has a free MemoryBuffer available then it is returned, otherwise a new one is created.

When the DataPool reaches the maximum number of buffers, it will not be able to create a new one. Instead, the DataPool picks a LogicalBuffer from the list and spills its corresponding MemoryBuffer to disk, giving it to the new LogicalBuffer. The former LogicalBuffer keeps a reference to the disk file, retrieving the data when an operation attempts to read its content. Finally, when a LogicalBuffer stops being used, its MemoryBuffer is put back to the DataPool. In order to minimize the number of data spills and recoveries, the DataPool chooses the victim LogicalBuffer using a priority-aware schema, which uses one queue for each operation type that creates the LogicalBuffers. These queues are ordered depending on the likelihood of the LogicalBuffers to be requested back from disk. For example, LogicalBuffers created in shuffle operations have the lowest priority, as they will have to wait until the end of the map phase to be merged. Therefore, they are the first candidates

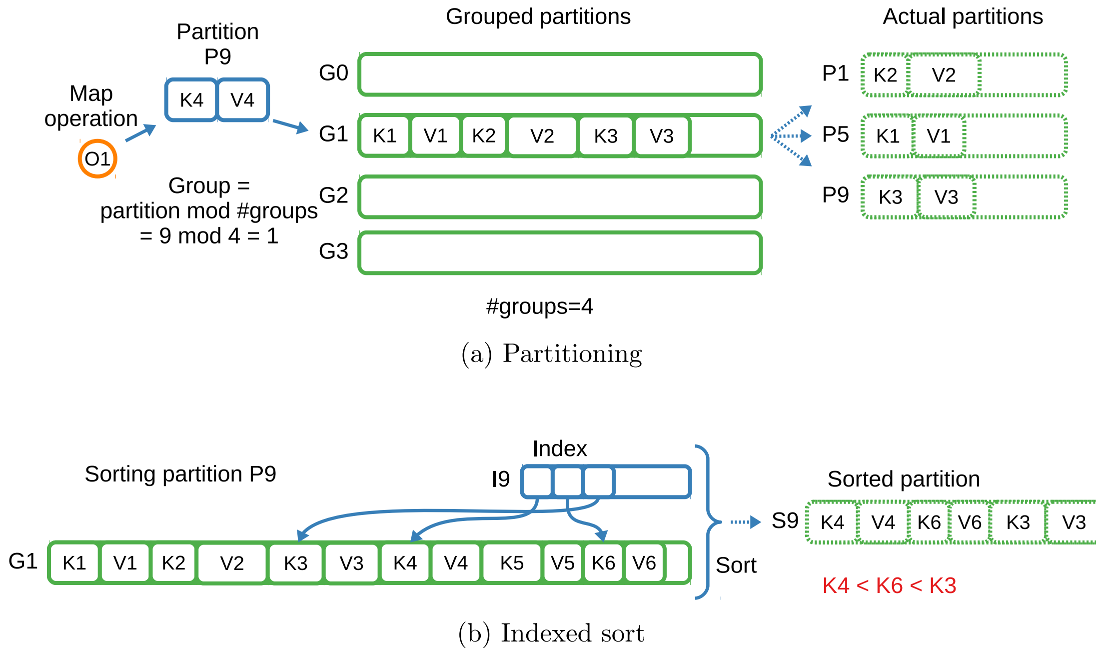
to be spilled.

In Flame-MR, most of the Worker memory is held by the DataPool, and so reusing MemoryBuffers reduces a high amount of memory allocations and deallocations. It also reduces the time taken to initialize the arrays when they are allocated, as the MemoryBuffers are reused without being reinitialized. Instead, they are lazily cleared by setting their writing position to zero. Moreover, both LogicalBuffers and MemoryBuffers use the same DataBuffer interface in order to keep a good design flexibility. MapReduce operations can therefore allocate buffers and operate the data without having specific information about where they are stored.

Buffered map output

One of the main objectives of Flame-MR is to minimize the redundant memory copies in operations like sort and merge. To this end, the key-value objects produced by the map operations are kept in memory until being sorted, without writing them to a temporary buffer. This behavior allows to reduce the memory copies but can lead to scalability issues with very large problem sizes due to the high amount of objects to be managed during garbage collection, causing long stalls. In order to overcome these limitations, the writing of the map output has been improved in Flame-MR-GCop by storing binary representations of the output pairs to a temporary buffer. This avoids to keep the objects in memory and reduces the garbage collection overhead, although it increases the number of memory copies. Furthermore, the partitioning and sorting mechanisms have been carefully studied, reassembling them for improved performance and scalability, as described next.

In Flame-MR-GCop, each mapper has a set of temporary buffers available for writing the output. When a map operation produces an output key-value pair, it is written to one of these buffers. Once a buffer is full, the output pairs are sorted and sent through the network. The number of times this event is triggered depends on how many temporary buffers are available for each mapper. Using one single buffer for all partitions is likely to cause an excessive number of sorts, as the buffer will be filled very quickly. However, having one buffer per partition is not a feasible option due to its poor scalability (the number of partitions increases with the number of computing nodes in the cluster). Therefore, the solution adopted in



(a) Partitioning

(b) Indexed sort

Figure 3.7: Map output example

Flame-MR-GCop is to arrange the partitions into a set of groups, calculated as a configurable percentage of the available memory. Each group has an assigned buffer, which contains the output pairs belonging to the partitions in that group. When this buffer is full, the sorter is in charge of iterating the several partitions of the group and sorting their corresponding pairs. Figure 3.7a shows an example of a pair partitioning, calculating the partition group number and its associated temporary buffer. The key-value pair $<\mathrm{K}_4,\mathrm{V}_4>$ is in partition 9, which belongs to group 1. This group contains those partitions P_i which meet the condition $i \bmod \#groups = 1$, which are P_1, P_5, P_9, and so on.

Each partition has an associated index that keeps track of the positions where the pairs are stored in the buffer. In the sort operation, this index is used for ordering the positions by comparing the pairs in the buffer. No data objects are created in this operation. Instead, the sort algorithm uses a binary comparison of the content of the positions. Figure 3.7b shows an example of a sort operation. The index (I_9) is used to access the positions that contain pairs $<\mathrm{K}_4,\mathrm{V}_4>$, $<\mathrm{K}_6,\mathrm{V}_6>$ and $<\mathrm{K}_3,\mathrm{V}_3>$. Then, the index is sorted according to the order of the keys, $\mathrm{K}_4 < \mathrm{K}_6 < \mathrm{K}_3$. Finally, the output pairs are copied to another buffer (S_9) which will be used for sending the

data to the corresponding Worker.

Experimental results

This section evaluates the optimizations described previously by comparing the original version of Flame-MR and the optimized one (Flame-MR-GCop) using the Sort benchmark. As the map and reduce functions do not perform any computation over the data, the results are only affected by the overall efficiency of the MapReduce engine, which makes Sort a suitable benchmark for evaluating memory optimizations. In order to avoid repetitive results, the optimizations have not been assessed separately, thus focusing on the benefits that Flame-MR-GCop can provide as a whole. Some early experiments have been carried out in order to assess the indexed sort mechanism, which reduced up to 39% the longest Garbage Collection Time (GCT) among the Workers compared to sorting the map output buffer by creating all the objects. This provided some reductions in execution time, and so the indexed sorting mechanism has been used in the next experiments. Regarding the number of map output buffers, by default they occupy up to 40% of the available Worker heap size. In the experiments, the corresponding number of buffers is higher than the total number of partitions in the cluster, so each buffer has been assigned to a single partition.

Table 3.3 shows the experimental results. The Execution Time (ET in the table) represents the time taken by the workload to be completed. Due to the high variability between experiments, the results include the minimum, median and maximum ET. The corresponding GCT metric also shows great variability, and so several results are provided. The best and worst cases represent the Workers with the shortest and longest GCT, respectively, while the total GCT represents the sum of the GCT from all the 32 Workers (i.e. 8 slave nodes and 4 Workers per node, see Table 3.2). The GCT of each Worker was obtained with jstat [118]. As can be seen, the memory management improvements of Flame-MR-GCop obtain a much lower GCT (best, worst and total) compared to the previous version. This is reflected on ET, especially in the maximum case, which has an 85% reduction in total GCT and a 44% reduction in ET. Note that the worst GCT is the value that determines the delay caused by GC overheads, since the fastest Workers will have to wait for the slowest ones. The results show a clear correlation between the worst GCT and ET

Table 3.3: Sort results for Flame-MR and Flame-MR-GCop
ET: Execution Time; GCT: Garbage Collection Time

Framework	ET Minimum	GCT Best / Worst / Total
Flame-MR	864s	10s / 34s / 685s
Flame-MR-GCop	649s	3s / 7s / 119s
	Median	**Best / Worst / Total**
Flame-MR	1000s	12s / 36s / 782s
Flame-MR-GCop	744s	3s / 6s / 128s
	Maximum	**Best / Worst / Total**
Flame-MR	1454s	14s / 170s / 828s
Flame-MR-GCop	819s	3s / 5s / 124s

for Flame-MR, but not for Flame-MR-GCop. This means that the worst GCT of the improved version does not have such a great impact on ET as in the previous version.

Figure 3.8 presents the evolution of memory usage over time. The graphs show two scenarios: the Worker with the shortest GCT (Figures 3.8a and 3.8b) and the longest GCT (Figures 3.8c and 3.8d) for the experiment with median ET, previously shown in Table 3.3. The lines in these graphs depict different values related with memory behavior, including the accumulated GCT along the execution of the Worker. The JVM memory size shows the memory occupied by the Worker process, while the heap usage shows the actual size of memory that is being used by Java objects. It can be seen that Flame-MR-GCop (Figures 3.8b and 3.8d) presents a significantly lower GCT than Flame-MR (Figures 3.8a and 3.8c), along with a more stable heap usage. In Flame-MR, the heap usage has a great variability over time, as the DataBuffers are created and dereferenced as needed. It can be observed that merge operations, which begin at 240s approximately in both scenarios, cause the highest variability in heap size, increasing the JVM size to its maximum value. This behavior is motivated by the high amount of DataBuffers that each merge operation consumes and produces, which also increases GCT, as shown in the graphs. In contrast, the heap usage of Flame-MR-GCop has a low variability, with increasing values until a certain maximum. Once this maximum is reached, the heap usage does not decrease,

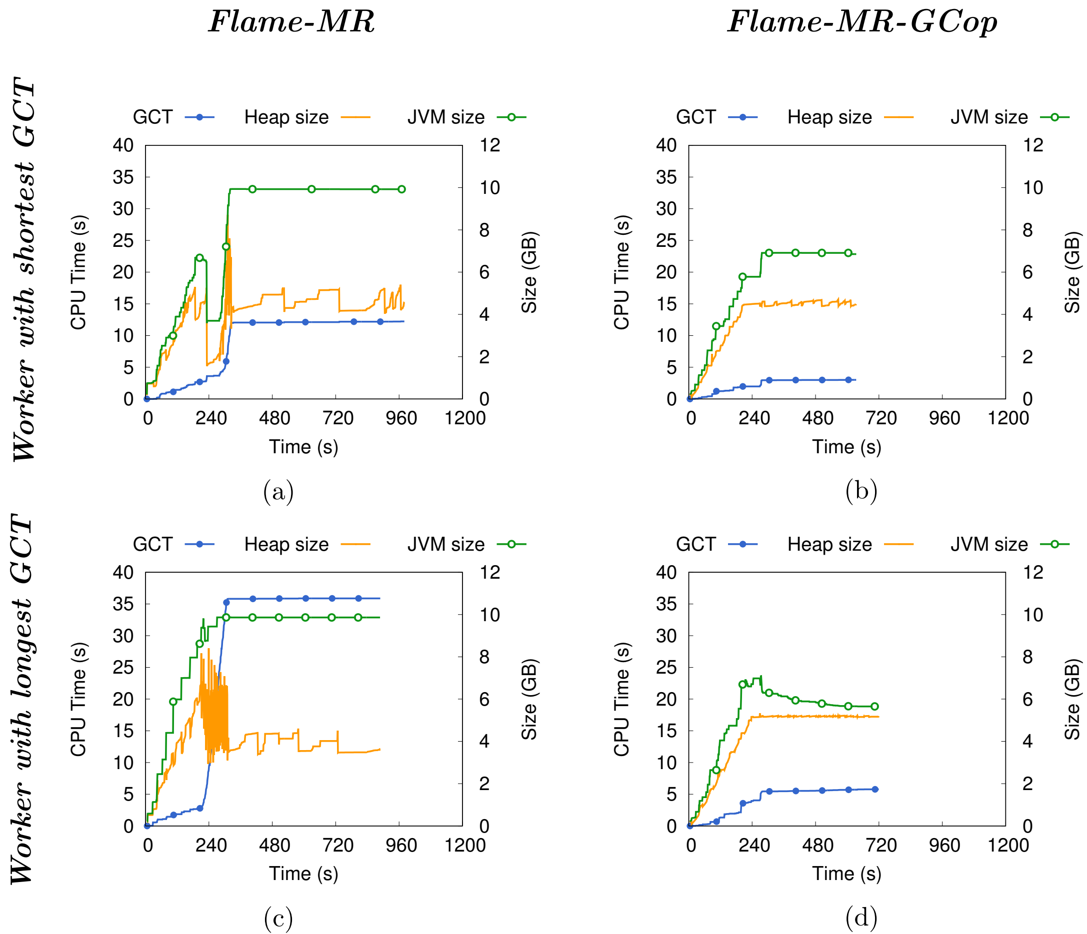


Figure 3.8: GCT and memory usage over time for Flame-MR and Flame-MR-GCop

as the DataBuffers are kept by the DataPool in order to be reused. The reuse of DataBuffers also reduces the GCT, making the merge operations go unnoticed in terms of memory activity compared with Flame-MR.

The results shown in this section demonstrate the impact of the GC overhead on workload performance, as well as the effectiveness of the static memory allocation and buffered map output techniques for reducing the GCT and thus the overall execution time. Moreover, the variability on heap size is also significantly reduced.

3.4.2. Buffer type analysis

In Java, there are two basic ways of allocating memory for new objects: on-heap and off-heap. On the one hand, on-heap memory is used by general objects and classes, being always tracked and deleted if needed by the GC. On the other hand, off-heap memory can be used for allocating buffers outside the Java heap (i.e. in native memory), which avoids copying data from heap space to native memory during OS calls (e.g. I/O operations). For storing bytes, which is the data type internally used by Flame-MR, both kinds of memory can be allocated using the ByteBuffer class provided by the JVM. ByteBuffer objects can then be used as the source and destination of I/O system calls. The underlying memory type that is being used is encapsulated by the ByteBuffer object, using the HeapByteBuffer subclass for on-heap memory (wrapping a primitive byte array) and the DirectByteBuffer subclass for off-heap memory (wrapping memory allocated outside the heap using a malloc-like call). Although off-heap memory is out of the control of the GC (i.e. memory buffers are never moved), DirectByteBuffer objects can be garbage collected. So, the deallocation of off-heap memory is also performed during GC execution. This section discusses the different alternatives for implementing MemoryBuffers, evaluating their performance and their impact on garbage collection.

MemoryBuffer implementations

As introduced in Section 3.3.1, Flame-MR manages memory using buffers (MemoryBuffer objects). The underlying implementation of these buffers is abstracted by the DataBuffer interface, and so their external behavior is separated from the actual memory operations they perform. This feature allows the implementation of several kinds of MemoryBuffers that allocate memory in different ways. By default, MemoryBuffers use primitive byte arrays to store the data in the heap. Additionally, two MemoryBuffer implementations have been developed in Flame-MR-GCop, using HeapByteBuffers and DirectByteBuffers, respectively.

On the one hand, HeapByteBuffers operate over encapsulated primitive byte arrays but also provide methods to extract them. This is useful to perform some operations in Flame-MR, like binary comparisons, by working directly with the byte array and overcoming certain limitations of the ByteBuffer interface. On the other

Table 3.4: Sort results for different buffer types in Flame-MR-GCop
ET: Execution Time; GCT: Garbage Collection Time

Buffer type	**ET** **Minimum**	**GCT** **Best / Worst / Total**
Byte array	649s	3s / 7s / 119s
HeapByteBuffer	681s	2s / 6s / 118s
DirectByteBuffer	649s	2s / 7s / 121s
	Median	**Best / Worst / Total**
Byte array	744s	3s / 6s / 128s
HeapByteBuffer	764s	3s / 5s / 117s
DirectByteBuffer	696s	2s / 11s / 130s
	Maximum	**Best / Worst / Total**
Byte array	819s	3s / 5s / 124s
HeapByteBuffer	855s	3s / 6s / 124s
DirectByteBuffer	779s	2s / 8s / 130s

hand, DirectByteBuffers can potentially reduce memory copies and improve the performance of some I/O operations. However, DirectByteBuffers do not provide any method to extract the underlying byte array, and so any data access must comply with the ByteBuffer interface. Moreover, Hadoop libraries do not currently support the use of ByteBuffers for writing data to HDFS, and so output operations performed by Flame-MR have to use primitive byte arrays. While HeapByteBuffers allow extracting the encapsulated arrays to perform these operations, DirectByteBuffers must first copy their data to the heap. These issues make it difficult to determine theoretically which MemoryBuffer implementation is the most suitable. The next section analyzes the performance of the different alternatives.

Experimental results

The experiments to evaluate the three implementations of MemoryBuffers were conducted in a similar way to those of Section 3.4.1. The ET and GCT results of the Sort executions are shown in Table 3.4. Regarding ET, it can be seen that

DirectByteBuffers obtain better results than the other two alternatives. Although the minimum ET is the same as with byte arrays, the median and maximum results are better. As mentioned before, HeapByteBuffers behave as encapsulated byte arrays, showing similar performance than these ones, plus the overhead of managing the actual ByteBuffer objects, which causes HeapByteBuffers to have a higher ET. Moreover, the results do not show any clear correlation between GCT and ET. Although DirectByteBuffers are the ones with slightly higher total GCT, the differences are almost negligible.

The graphs of Figure 3.9 show the accumulated GCT, heap size and JVM size of the Worker with the shortest and the longest GCT for the experiment with median ET (see Table 3.4). As expected, these graphs do not show any significant variation between byte arrays (Figures 3.9a and 3.9d) and HeapByteBuffers (Figures 3.9b and 3.9e), as they are using the same kind of memory in the end. However, the behavior of DirectByteBuffers (Figures 3.9c and 3.9f) greatly differs from them, showing a smaller heap size and larger JVM size. The former is caused by the off-heap allocation of DirectByteBuffers, which is obviously not reflected in the heap size. Meanwhile, the additional copies needed for accessing the data from the heap are responsible for the larger JVM size. The use of off-heap memory also reduces the GCT during most of the computation time. However, the scenario of the Worker with the longest GCT (Figure 3.9f) shows a remarkable characteristic, which is a high amount of GCT consumed in a short period of time when reaching the end of the workload. This is caused by the deallocation of off-heap buffers to make room for on-heap ones, needed to write the final output to HDFS due to its API incompatibility. When the off-heap MemoryBuffers are deleted from the DataPool, the DirectByteBuffers stop being referenced, and so the GC collects them. The deallocation of off-heap memory, included in this process, incurs a high performance penalty, and so the GCT is affected.

The conclusion that can be extracted from these results is that off-heap buffers generally obtain a better performance than on-heap alternatives. Although some operations are more expensive when performed outside the heap (e.g. binary comparison), the improvement of I/O operations allows to get better overall results. Currently, the main issue about using DirectByteBuffers is the lack of an API method for storing them to HDFS, which is expected to be overcome with future releases of

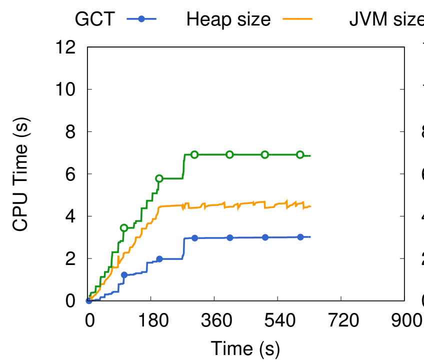
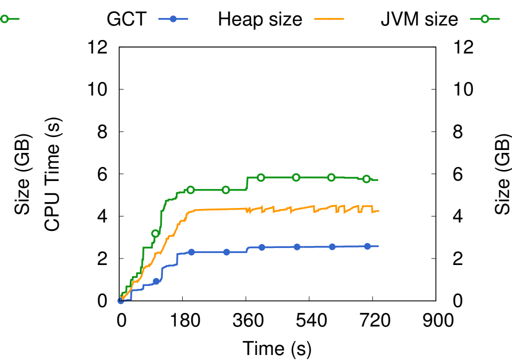
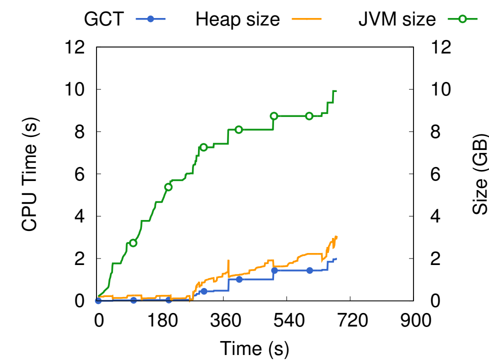
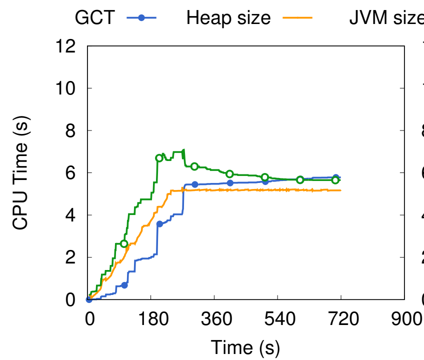
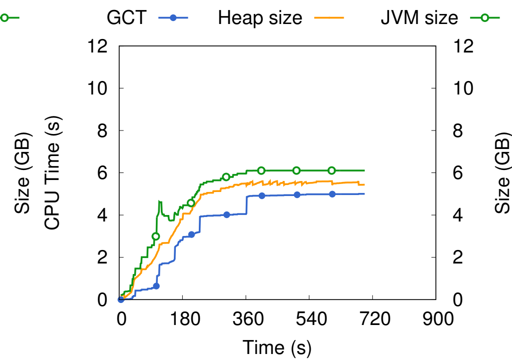
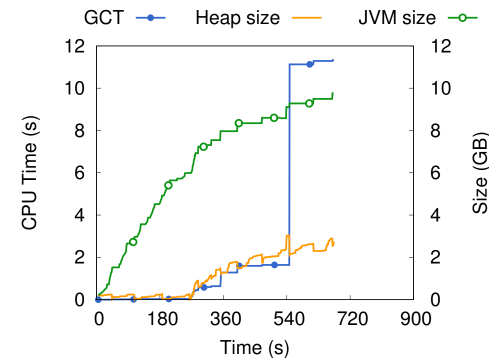
	Byte array	*HeapByteBuffer*	*DirectByteBuffer*
Worker with shortest GCT	 (a)	 (b)	 (c)
Worker with longest GCT	 (d)	 (e)	 (f)

Figure 3.9: GCT and memory usage over time for the different DataBuffer implementations in Flame-MR-GCop

Hadoop libraries.

3.4.3. Iterative support

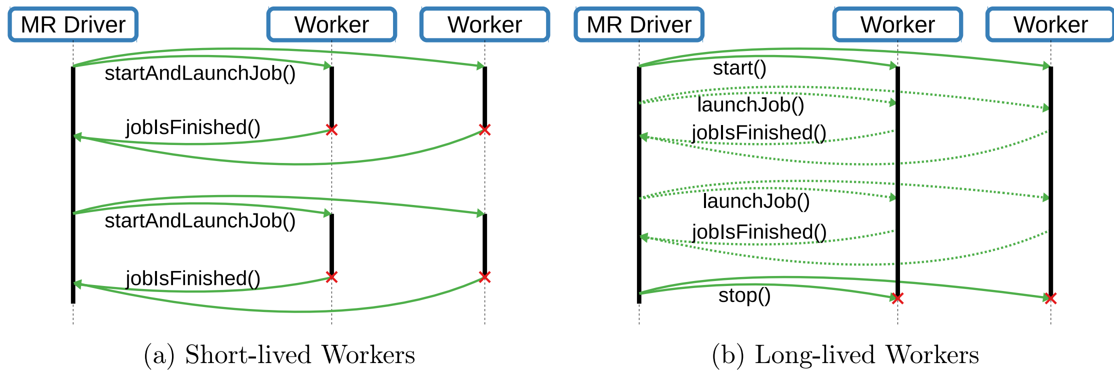
In real use cases, most MapReduce applications are composed by more than a single job, iterating several times over the input dataset to get a final result. Each MapReduce job reads the output of the previous one and generates a new dataset, until meeting a certain stop condition or reaching the maximum number of iterations. Although iterative applications are very common, MapReduce frameworks like Hadoop are not well suited for them. This is caused by the writing of intermediate results to HDFS and the high overhead of launching and finishing jobs, which leads to poor performance of the overall workload.

Like Hadoop, the versions of Flame-MR discussed in the previous sections are not oriented to iterative applications. This section focuses on how to improve the performance of these applications, avoiding to write intermediate results to HDFS by caching data in memory to minimize the use of disk and maximize in-memory processing. Two steps have been carried out to achieve this objective: the use of long-lived Workers and the implementation of a data cache.

Long-lived Workers

In each execution of a MapReduce job, Flame-MR starts one or more Workers per computing node in the cluster. When the job finishes these Workers are stopped, and hence they have to be restarted to launch the following job. In a similar way, Hadoop starts a Java process for each mapper or reducer, and so each job also involves the launching and stopping of multiple JVMs. Both approaches prevent caching intermediate results in memory between iterations, as all in-memory data are lost when the corresponding JVMs are finalized (i.e. all the output data have to be written to HDFS). Therefore, the first requisite for implementing a data cache in Flame-MR is to avoid stopping and restarting the Workers, reusing them through all the MapReduce jobs performed during the execution of an application. This feature also minimizes the overhead of launching new Worker processes for each job.

The proposed solution is to modify the way Workers are managed, avoiding the



(a) Short-lived Workers (b) Long-lived Workers

Figure 3.10: Short-lived vs long-lived Workers

finalization of their JVMs when a MapReduce job is completed. Figure 3.10 depicts the main differences between the previous behavior of Workers, called short-lived Workers, and the new approach, long-lived Workers. On the one hand, short-lived Workers (see Figure 3.10a) are started for executing a certain job, being stopped after this job is finished. On the other hand, long-lived Workers (Figure 3.10b) are reused several times. Instead of being started for running a job, these Workers are initialized at the beginning of the MapReduce Driver (MR Driver in the figures). They keep waiting until they receive a job launch message from the driver and execute the corresponding MapReduce job. When the job finishes, they send a message back to the driver and keep waiting for more jobs to run. When the application concludes, the driver sends a message for stopping the Workers and releasing the computational resources allocated to them.

The use of long-lived Workers is mandatory to implement a data cache. It also improves the performance of iterative applications, reducing the overhead between iterations by avoiding the costly initialization of new JVM processes in each job. Furthermore, the internal structures of Flame-MR, such as DataPool and Thread-Pool, are also reused along the entire application workflow, initializing them only once. Note that this includes the costly allocation and initialization of Memory-Buffers, already commented in Section 3.4.1.

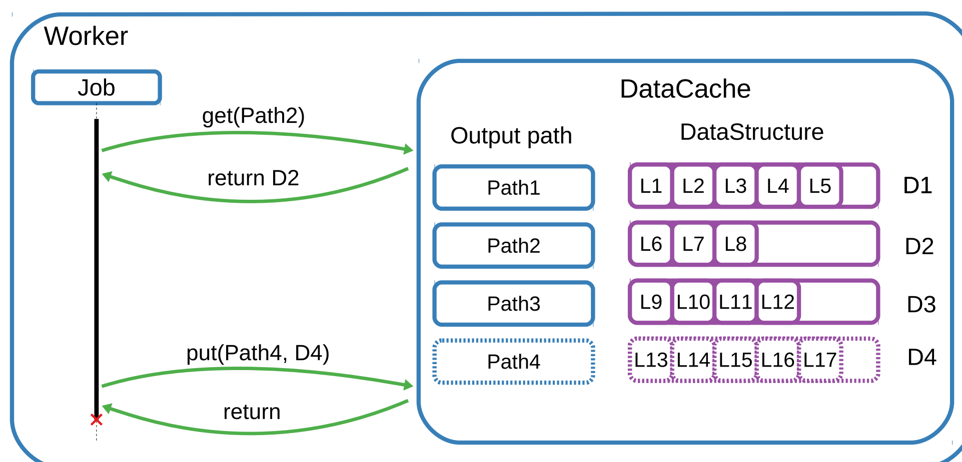


Figure 3.11: Data cache example

Data cache

Long-lived Workers can be configured to use a data cache in order to avoid the writing of intermediate results to HDFS. When this feature is activated, each Worker stores its output DataBuffers to an in-memory DataStructure, which is indexed in the DataCache. Figure 3.11 illustrates the behavior of the DataCache with an example of a cache hit. The output from several MapReduce jobs is tracked by associating the output path of each one (Path_i) with its corresponding DataStructure (D_i), which contains a set of LogicalBuffers (L_j). When a new job starts, the Worker reads the entries in the DataCache to check if the input path of the job is present. In this example, the input data (D_2) is read directly from the DataStructure stored in the DataCache. Otherwise, the input data has to be read from HDFS. When the job finishes, it adds its own output DataStructure (D_4) to the DataCache, making it available for future jobs. When the entire application finishes and the Worker is stopped, the contents of the DataCache are flushed to HDFS making use of the path information. After that, the dataset stored in HDFS is equivalent to the one written by a non-cached execution of the application.

It is important to note that the output of a MapReduce job is often accessed by the MR Driver before the entire application is completed. In fact, many iterative applications perform different kinds of operations over the intermediate results, like moving data from one path to another, deleting discarded results or reading some

of them to check a stop condition. In these cases, the data cached by the Workers would be unavailable for the MR Driver, causing either errors in the driver program or incorrect results. Thus, the behavior of HDFS calls must be modified in order to ensure the same results as in non-cached jobs. First, jobs create empty output directories when finished, so the HDFS state from the point of view of the MR Driver remains the same. The HDFS calls are then monitored during execution to modify the content of the DataCaches accordingly. When the MR Driver attempts to delete a path that is cached, its entry is removed and the DataBuffers are released. When a path is moved, the corresponding entries change the associated path, with no writings to HDFS. Finally, when the MR Driver attempts to read a dataset, the Workers flush the content of the cached DataStructures to HDFS, filling the empty output directories. After the data are available in HDFS, the corresponding DataCache entries are removed and the MR Driver can then access data in the standard way. This behavior ensures that the data of the output directory are always entirely in memory or in HDFS, preventing any inconsistency in subsequent jobs. However, in some cases, a MapReduce job can process data from HDFS and the DataCache simultaneously, when the input is composed of several paths stored in different places. As data are independently stored either in DataBuffers or HDFS blocks, they can always be read and processed in parallel. Note also that HDFS generally keeps several replicas of each data block as configured by the user, which ensures the reliability of the intermediate results. Our current implementation of the DataCache does not perform any replication, and so if a Worker is torn down, it would mean the loss of all its intermediate results. Although the reliability of the final results is ensured after writing them to HDFS, new fault tolerance mechanisms are needed to avoid data loss in case of Worker breakdown, but reducing disk and network overheads.

The approach described has some similarities with the one implemented in the M3R framework [107], which uses a key-value pair cache through the execution of in-memory iterative jobs. However, our implementation differs in the way data are represented in memory, using instead a binary format to store the data in DataBuffers and thus reducing the overhead incurred by the GC tracking. Futhermore, the data size can be higher than the available memory space, in which case Flame-MR will spill some of the buffers to disk, while M3R can only work with in-memory data and thus with a maximum data size.

Table 3.5: Execution times for PageRank

Flame-MR	**Flame-MR-GCop**	**Flame-MR-It-NoCache**	**Flame-MR-It-Cache**
1895s	1429s	1176s	1060s

The optimizations explained in this section, long-lived Workers and the data cache, have been integrated in Flame-MR-GCop, resulting in a new version called Flame-MR-It. The following section analyzes the performance improvement obtained with this iterative-aware version.

Experimental results

This section analyzes the performance and resource efficiency of the iterative application PageRank. Although other iterative workloads have also been tested, the results did not differ significantly from the ones obtained with PageRank, and so they were not included in this section. Table 3.5 shows the median execution time of the different versions of Flame-MR. In order to analyze the performance improvement obtained by the use of the data cache, Flame-MR-It has been executed with and without activating the cache (Flame-MR-It-Cache and Flame-MR-It-NoCache, respectively). The results show that both configurations of Flame-MR-It reduce significantly the execution time of PageRank with respect to the previous versions. Using Flame-MR-GCop as baseline, Flame-MR-It-NoCache reduces the execution time by 18%, while Flame-MR-It-Cache decreases it by 26%. Hence, most part of the obtained improvement is because of the use of long-lived Workers. This reveals that for this application the initialization of the Workers has a significant impact on performance, being even more significant than the writing of the intermediate results to HDFS (only 38 GB per iteration in this case), avoided by the use of the data cache.

Figure 3.12 depicts the resource utilization statistics of Flame-MR and both versions of Flame-MR-It. In terms of CPU usage, both Flame-MR-It versions show lower values along the entire computation of the workload. This is caused by the GC optimizations explained in Section 3.4.1, as well as by the reduction of the initialization overhead between iterations. Furthermore, the activation of the data

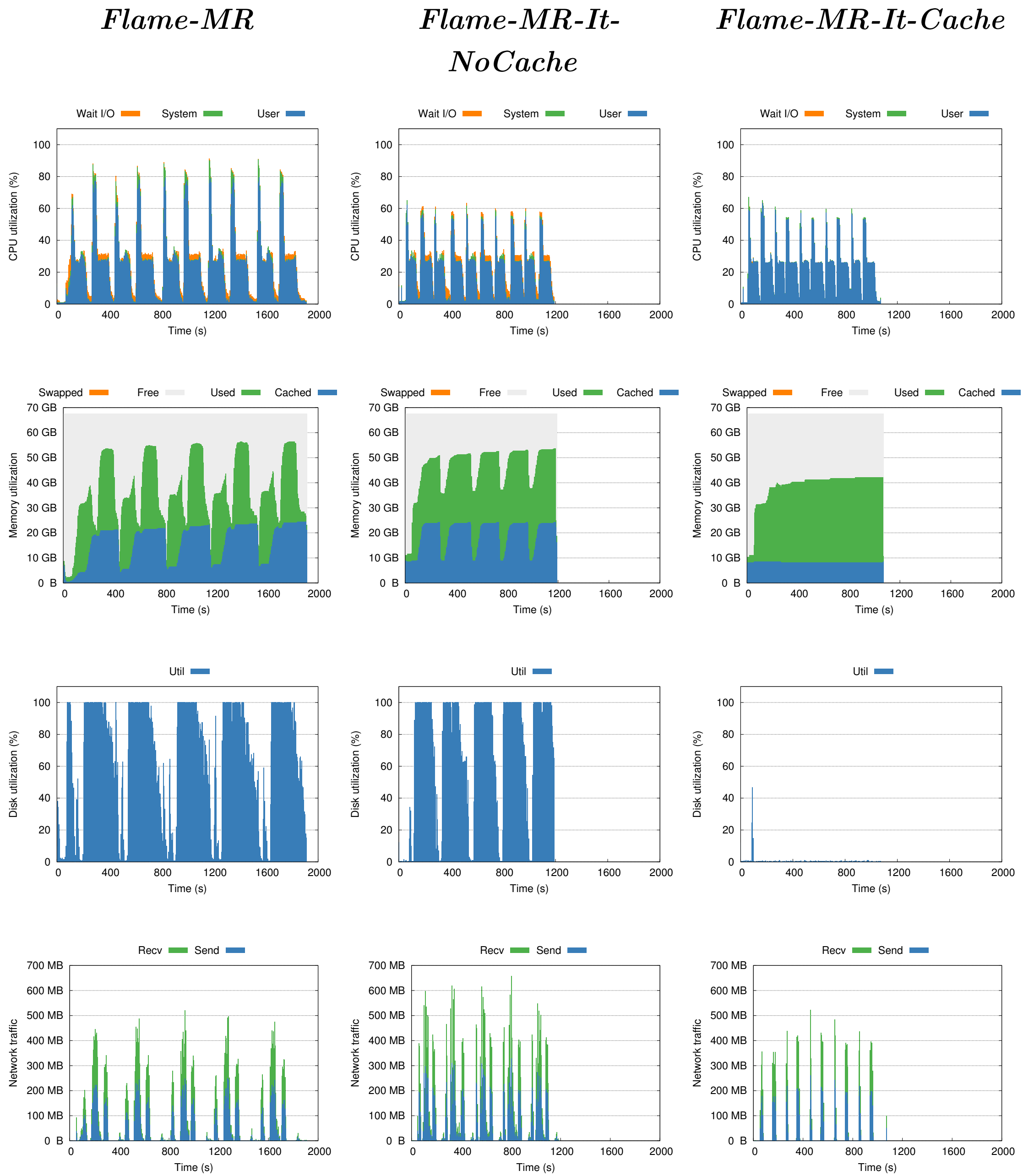


Figure 3.12: Resource utilization statistics of Flame-MR, Flame-MR-It-NoCache and Flame-MR-It-Cache

cache has a direct effect on CPU usage, almost eliminating the CPU time waiting for I/O. Regarding memory usage[1], Flame-MR-It-NoCache takes further advantage of the memory optimizations, avoiding the maximum and minimum peaks of Flame-MR by means of the static DataBuffer allocation (see Section 3.4.1). The use of long-lived Workers also improves memory usage by avoiding the release of the entire JVM memory space each time a job is finished. Activating the data cache also contributes to a more stable memory usage, as it avoids the increase of the memory used by the OS buffer cache.

Both Flame-MR and Flame-MR-It-NoCache show high disk utilization in each iteration of PageRank, reaching 100% values during several intervals of the workload. Meanwhile, Flame-MR-It-Cache shows almost no disk utilization, as the intermediate results are neither written to nor read from HDFS. Regarding network traffic, both Flame-MR and Flame-MR-It-NoCache show higher values than Flame-MR-It-Cache. This is caused by the sending of data blocks through the network when they are replicated according to the configured replication factor (3, see Table 3.2). As Flame-MR-It-Cache does not write any intermediate results to HDFS, it only incurs the traffic belonging to the shuffle phase. In conclusion, the use of the data cache not only improves performance but also increases resource efficiency.

3.5. Conclusions

Hadoop is the most popular open-source MapReduce framework to handle Big Data applications. Although the need for improving the performance of MapReduce applications is steadily increasing, the high cost (or impossibility) of rewriting their source code can make the adoption of new frameworks like Spark or Flink unfeasible. In order to overcome this situation, this chapter has presented Flame-MR, a new MapReduce framework that improves the performance of Hadoop without modifying the source code of the applications. Flame-MR transparently replaces the internal design of Hadoop with an event-driven architecture that optimizes the use of memory and CPU resources, while also alleviating other performance bottlenecks such as redundant memory copies and the overhead of object creation/destruction. Fur-

[1]Note that the label "Cached" refers to the OS buffer cache, not to the data cache of Flame-MR-It-Cache

thermore, it pipelines the output of map and reduce phases to decrease disk usage and improve the overlapping of data processing with disk and network operations.

Future systems are expected to have increasing memory sizes, which can be challenging to current data processing frameworks. Thus, the impact of memory efficiency on the performance of Flame-MR has been analyzed in depth, presenting several memory optimization techniques that have been implemented and evaluated. The obtained results have shown that these techniques can reduce the amount of object allocations and deallocations, decreasing GC overheads and overall execution times by 85% and 44%, respectively. Moreover, several memory buffer implementations have been analyzed, showing that direct byte buffers can improve the performance of I/O-bound operations. Finally, the performance of iterative workloads was improved by reusing Worker processes and caching intermediate data to avoid unnecessary writes to HDFS, reducing execution times by 26%.

Chapter 4

Experimental analysis of Flame-MR in cluster and cloud platforms

The previous chapter described the design and implementation of Flame-MR, as well as some optimizations regarding the efficient use of memory resources for large-scale data processing. This chapter evaluates the performance of Flame-MR at large scale using different scenarios. First, the performance of Flame-MR is compared with Hadoop and Spark when executing standard benchmarks in the Amazon EC2 cloud platform [1]. The experimental results show significant performance benefits reducing Hadoop execution times by up to 65%, while providing very competitive results compared to Spark. Second, the performance improvement of real-world use cases is analyzed by evaluating three MapReduce applications that are commonly used to process large-scale datasets in different research areas. The experimental evaluation, conducted in high performance clusters and the Microsoft Azure cloud [88], shows a clear outperformance of Flame-MR over Hadoop. In most cases, Flame-MR reduces the execution times by more than a half.

The chapter is organized as follows. Section 4.1 describes related works that evaluate the performance obtained by framework optimizations. Section 4.2 addresses the evaluation of Flame-MR using standard benchmarks, while Section 4.3 analyzes three different real-world use cases by performing an experimental evaluation for each of them. Finally, Section 4.4 provides the main conclusions extracted from the results.

4.1. Related work

As commented in Section 3.2, many papers in the literature have compared the performance benefits of using the MapReduce model with different data processing engines (e.g. NativeTask [141]), file systems (e.g. MARIANE [36]), and network interconnects (e.g. RDMA-Hadoop [132]). These papers generally execute popular Big Data benchmarks like TeraSort or K-Means. The use of these benchmarks allows to compare more easily the benefits of the different optimizations, although they are affected by the experimental testbed that is being used in each specific case. However, the sole use of these results makes it difficult to determine the actual performance benefit that a user can obtain when replacing Hadoop with any of the optimized alternatives.

Regarding large-scale applications employed in real use cases, their optimization is often performed by translating their source code to a more efficient computing paradigm. For example, Kira [148] is a distributed astronomy image processing toolkit implemented on top of Spark. It can obtain a 3.7× speedup on the Amazon EC2 cloud over an equivalent parallel implementation written in C and running on the GlusterFS file system. A similar approach has been employed in [55] to adapt high energy physics workflows to Spark, obtaining improved usability and performance when compared to other existing sequential implementations like ROOT [18]. Although these works prove to accelerate the execution of real-world applications, a considerable effort is required to translate existing applications and libraries to a new computing paradigm.

Some other works use these applications to determine the performance benefits of framework optimizations. For example, the Kira toolkit is used in [116] to evaluate RDMA-Spark [79], which improves the results of standard Spark with a 1.21× speedup. In the case of Hadoop, the authors of OEHadoop [115] evaluate their proposal by simulating a Facebook job trace extracted from the SWIM project [22]. OEHadoop, which offloads data replication to a low-level optical multicast system, obtains better performance than the original Hadoop, although the results provided are extracted from simulations and not from empirical data.

One of the most important requirements that framework optimizations must meet is portability, as the same MapReduce application is likely to be executed

in many different systems. This makes Flame-MR a good candidate to improve performance by leveraging memory resources, as it has been specifically designed to accelerate applications in a transparent and portable way.

4.2. Performance comparison with Hadoop and Spark in the cloud

This section presents an experimental evaluation of Flame-MR in the cloud compared to popular Big Data processing frameworks. The goal is to obtain performance evaluation results of the original version of Flame-MR described in Section 3.3, as well as to determine the benefits of the memory-enhanced and iterative-aware versions described in Section 3.4 (i.e. Flame-MR-It).

First, Section 4.2.1 compares Flame-MR-It with the de-facto standard MapReduce implementation, Hadoop [8], assessing the impact of managing memory in a more efficient way when executing representative MapReduce workloads at a sufficiently large scale. This evaluation takes into account both performance and resource utilization metrics, also including the results for the original version of Flame-MR in order to determine the improvement achieved by all the optimizations presented in Section 3.4. Then, Section 4.2.2 analyzes the performance benefits of using Flame-MR-It compared to a state-of-the-art in-memory framework, Spark [145]. Note that both frameworks are oriented to improve the performance of Hadoop. Hence, comparing them is useful to determine the performance gain that Flame-MR-It provides, unlike Spark, without modifying the source code of existing MapReduce applications.

The experiments have been carried out in a public cloud infrastructure, Amazon EC2 [1], which is the most popular Infrastructure as a Service (IaaS) platform. In order to evaluate the behavior of the frameworks under different memory configurations, two Amazon EC2 instance types have been used: c3.4xlarge and i2.4xlarge, which were allocated in the US East (North Virginia) region. Table 4.1 contains the main hardware and software characteristics of both instances, as advertised by Amazon. As can be seen, the memory size of c3.4xlarge is 30 GB, while i2.4xlarge has 122 GB. The wide difference in terms of memory between these instances is of great interest to study the performance of the frameworks when running in a

Table 4.1: Node characteristics of Amazon EC2 instances

Hardware configuration

	c3.4xlarge	**i2.4xlarge**
CPU model	2 × Intel Xeon E5-2680 v2 (Ivy Bridge-EP)	2 × Intel Xeon E5-2670 v2 (Ivy Bridge-EP)
CPU Speed (Turbo)	2.80 GHz (3.60 GHz)	2.50 GHz (3.30 GHz)
#Cores	16	16
Cache (L1 / L2 / L3)	32 KB / 256 KB / 25 MB	32 KB / 256 KB / 25 MB
Memory	30 GB DDR3 1600 MHz	122 GB DDR3 1600 MHz
Disk	2 × 160 GB SSD	4 × 800 GB SSD
Network	Gigabit Ethernet	Gigabit Ethernet

Software configuration

	c3.4xlarge / i2.4xlarge
OS version	Amazon Linux AMI 2016.09
Kernel	4.4.19-29.55
Java	OpenJDK 1.7.0_111
Scala	2.10.6

more memory-constrained system (i.e. c3.4xlarge) compared to i2.4xlarge instances, allowing to analyze the impact of memory management under different system configurations. Furthermore, i2.4xlarge instances also provide more local storage: 4 SSD disks of 800 GB vs. 2 SSD disks of 160 GB for c3.4xlarge. However, note that the CPU speed of c3.4xlarge is slightly higher, which may improve the performance of CPU-bound workloads. The experiments were run on a 33-instance cluster (i.e. 1 master and 32 slaves).

Several benchmarks have been analyzed by using BDEv: WordCount, Sort, PageRank, Connected Components and K-Means. In the experiments, WordCount and Sort processed a 500 GB dataset, while PageRank and Connected Components performed 5 iterations over a 40 GB dataset (60 Mpages). K-Means processed an input dataset of 130 GB (3 Gsamples), using 10 clusters and performing a maximum of 5 iterations. The metric shown in the graphs is the mean value of 10 measurements

for each experiment, clearing the OS buffer cache between each execution.

Finally, the evaluated frameworks have been Hadoop 2.7.2, Spark 1.6.3, the original version of Flame-MR and its optimized version, Flame-MR-It. The latter has been configured to use the data cache in the iterative benchmarks (PageRank, Connected Components and K-Means), and byte arrays as the underlying implementation for DataBuffers. In order to provide the best results for each framework, their configuration, shown in Tables 4.2 and 4.3, has been carefully adapted to the characteristics of the instances, adjusting some of the parameters (e.g. DataBuffer size) by experimental tuning.

4.2.1. Comparison with Hadoop

Figures 4.1a and 4.1b show the execution times when using c3.4xlarge and i2.4xlarge instances, respectively. Note that the results of K-Means with Flame-MR are not included as the original version did not provide support for Mahout workloads. As can be observed, Flame-MR-It significantly outperforms Hadoop with an average improvement of 32% and 48% in c3.4xlarge and i2.4xlarge, respectively. These results confirm that Flame-MR-It presents a significantly better performance than Hadoop, especially when using large memory sizes as the improvements are higher as the memory of the instances increases. The only case in which Hadoop obtains a similar performance to Flame-MR-It is when running WordCount in c3.4xlarge. This is caused by the high amount of Java objects created by the user-defined map function, which affects the performance in memory-constrained systems, as the GC has to perform a lot of collections to make room for new objects. The results also show the significant performance improvement of Flame-MR-It over Flame-MR, demonstrating the effectiveness of the in-memory techniques developed.

Note that the choice of the instance type has a significant impact on the performance of the frameworks. The benchmarks that are influenced the most are WordCount and Sort. In fact, Hadoop and Flame-MR obtain lower execution times for WordCount in c3.4xlarge than in i2.4xlarge. This means that, for this benchmark, both frameworks are clearly constrained by the CPU power, and the larger memory and better disk features of i2.4xlarge do not compensate for its lower CPU speed (see Table 4.1). However, Flame-MR-It is not affected by this problem, achieving

Table 4.2: Configuration of Hadoop and HDFS in Amazon EC2

Hadoop

	c3.4xlarge	**i2.4xlarge**
Mappers per node	8	8
Reducers per node	8	8
Mapper/Reducer heap size	1.7 GB	6.33 GB
IO sort MB	380 MB	1.58 GB
Shuffle parallel copies	20	20
IO sort spill percent	80%	80%

HDFS

	c3.4xlarge / i2.4xlarge
HDFS block size	128 MB
Replication factor	3

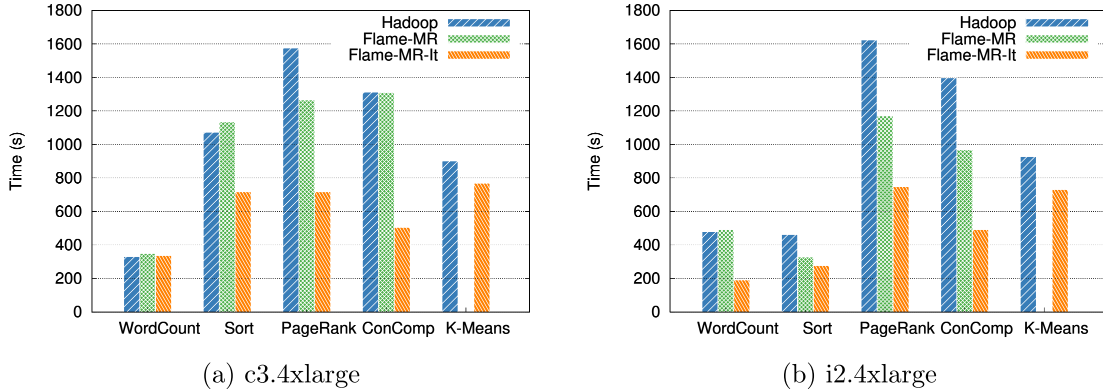
Table 4.3: Configuration of Flame-MR and Spark in Amazon EC2

Flame-MR / Flame-MR-It

	c3.4xlarge	**i2.4xlarge**
Workers per node	4	4
ThreadPool size	4	4
Worker heap size	5.4 GB	22.8 GB
DataPool size	3.3 GB	15.2 GB
DataBuffer size	512 KB / 128 KB	1 MB / 256 KB

Spark

	c3.4xlarge	**i2.4xlarge**
Executor heap size	23.8 GB	101.3 GB
Executors per node	1	1
Executor cores	16	16



(a) c3.4xlarge (b) i2.4xlarge

Figure 4.1: Execution times of Hadoop, Flame-MR and Flame-MR-It in Amazon EC2

for WordCount better performance in i2.4xlarge than in c3.4xlarge. This indicates that Flame-MR-It obtains a higher benefit than Hadoop from the use of large memory sizes, without being penalized by the CPU speed. Regarding Sort, it is the benchmark that shows the most important benefit from using i2.4xlarge. Sort is an I/O-bound workload, so this improvement can be clearly attributed to the higher number of disks provided by i2.4xlarge (4 disks), which improves the bandwidth of I/O operations, along with more memory space (122 GB), allowing the frameworks to retain more data in memory without spilling them to disk. As in the case of WordCount, Flame-MR-It benefits more than Hadoop from the use of i2.4xlarge instances, which means that the Flame-MR architecture together with the in-memory techniques presented in Section 3.4.1 are more suited to systems with larger memory sizes.

Finally, PageRank, Connected Components (ConComp in the figures) and K-Means are iterative applications, and so their performance is determined by the behavior within each iteration and between iterations. Hence, Flame-MR-It not only improves the execution time of each iteration due to its memory optimizations, but also reduces the overhead between iterations (e.g. launching of Worker processes) by means of the techniques explained in Section 3.4.3: reusing the Worker processes and avoiding to write intermediate results to HDFS by caching data in memory.

The resource utilization of Hadoop and Flame-MR-It can be useful to analyze their behavior in the c3.4xlarge and i2.4xlarge instances. As previously commented,

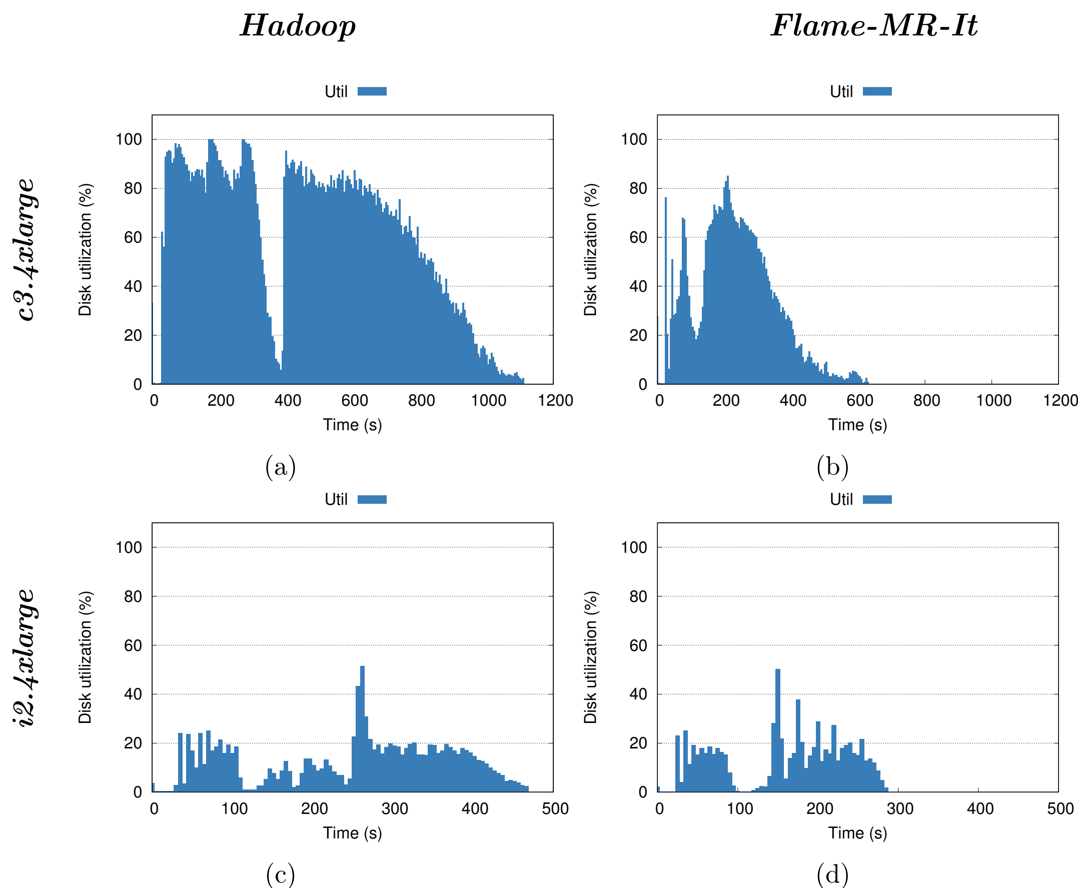


Figure 4.2: Disk utilization of Hadoop and Flame-MR-It for Sort

Sort was the benchmark which showed wider performance differences between both instance types. Therefore, it has been selected for analyzing the resource utilization of both frameworks, calculated as the average values among the slave nodes during the experiment with the median execution time. The disk and memory utilization results are shown in Figures 4.2 and 4.3, respectively.

As can be observed in Figure 4.2, the disk constitutes the main performance bottleneck in c3.4xlarge (see Figures 4.2a and 4.2b), while the larger memory size and higher number of disks provided by i2.4xlarge alleviate the load per disk (Figures 4.2c and 4.2d). This, in turn, accelerates the completion of the benchmark by 2.32× and 2.60× for Hadoop and Flame-MR-It, respectively. In c3.4xlarge, Flame-MR-It presents lower disk utilization than Hadoop along the entire execution thanks to its better in-memory computing capabilities.

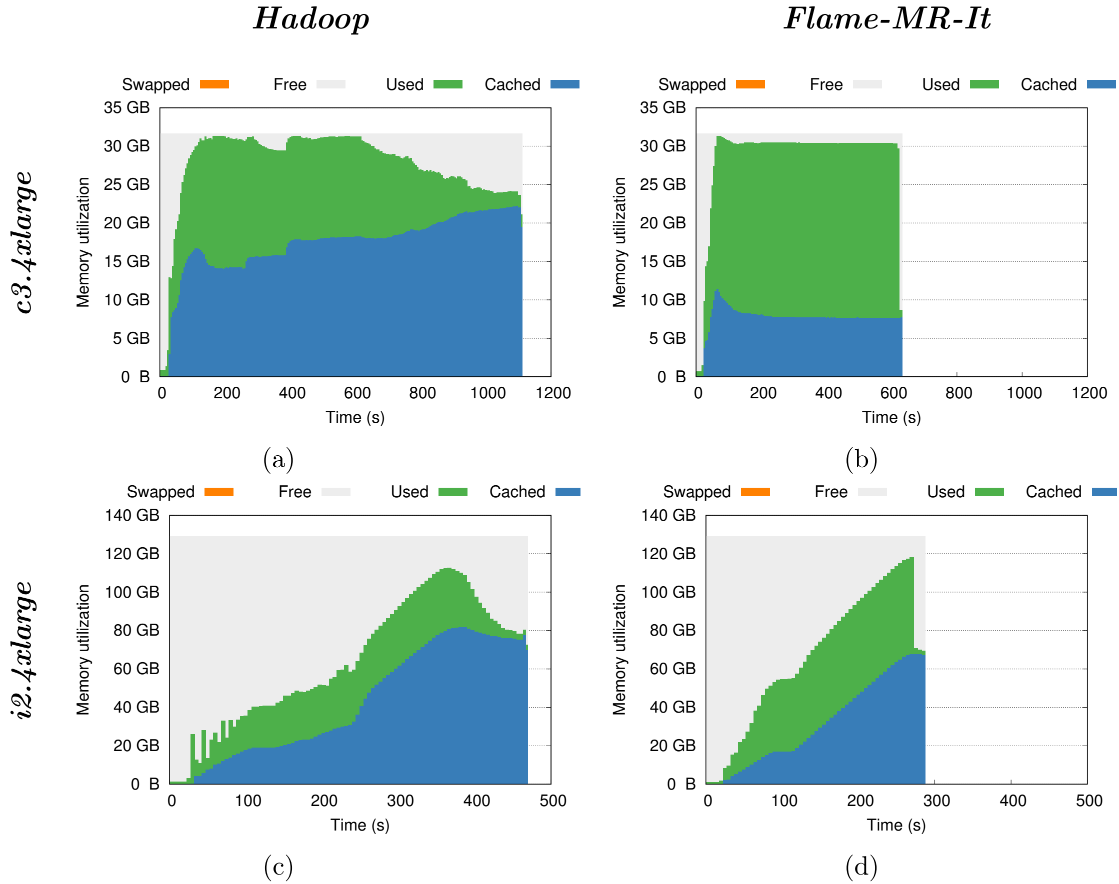
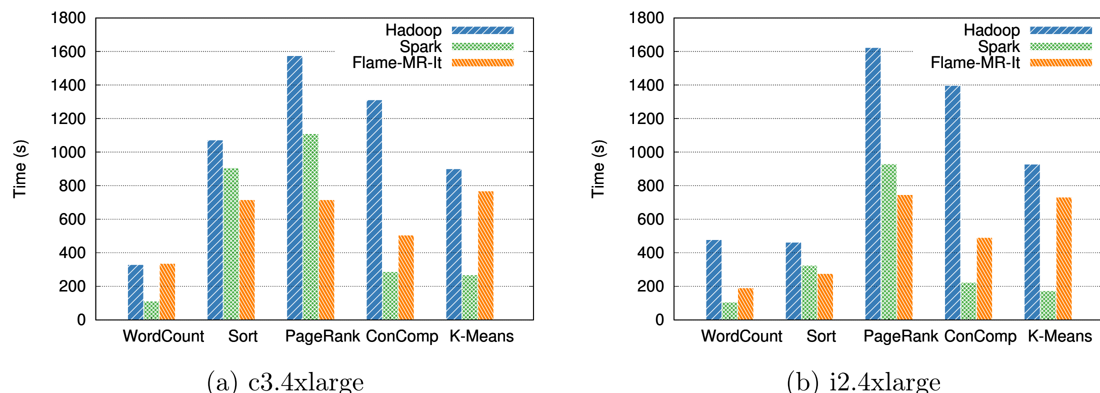


Figure 4.3: Memory utilization of Hadoop and Flame-MR-It for Sort

Regarding memory utilization, Figures 4.3a and 4.3c show that Hadoop heavily relies on the OS buffer cache to accelerate I/O operations, which matches with the high disk utilization values commented before. Meanwhile, Flame-MR-It (Figures 4.3b and 4.3d) keeps lower buffer cache values by avoiding disk access as much as possible and retaining more data in the Workers' memory. This feature allows Flame-MR-It to have more memory space available to be used even in memory-constrained systems like c3.4xlarge instances.

4.2.2. Comparison with Spark

Figure 4.4 shows the performance results of Hadoop, Spark and Flame-MR-It. As can be seen, the best performer depends on the workload being executed. On the



(a) c3.4xlarge

(b) i2.4xlarge

Figure 4.4: Execution times of Hadoop, Spark and Flame-MR-It in Amazon EC2

one hand, WordCount obtains the lowest execution times with Spark. This is due to the better suitability of the in-memory data structures of Spark to CPU-bound workloads like this one, combined with the use of efficient data processing operators like the *reduceByKey()* function. On the other hand, Flame-MR-It obtains the best results for Sort, mainly because of the great data sorting capabilities of the MapReduce model together with the optimized memory and GC management for large datasets provided by Flame-MR-It. Regarding iterative workloads, PageRank, Connected Components and K-Means also differ on the best framework to use. The in-memory optimizations of Flame-MR-It prove to be highly efficient for PageRank, reducing the execution time of Spark by 35% and 20% in c3.4xlarge and i2.4xlarge, respectively. However, Spark obtains better performance than Flame-MR-It for Connected Components and K-Means. This is due to the use of its built-in graph processing engine, GraphX, and its machine learning library, MLlib, that fully leverage the in-memory data operators of Spark. Note that the Spark implementation of these iterative workloads can use in-memory data to check convergence at the end of each iteration, whereas the Hadoop implementation used by Flame-MR-It must read the data from HDFS to perform this task, thus incurring additional network and disk overhead in the MR Driver.

In order to provide more information about the performance differences of Spark and Flame-MR-It, their resource utilization has been analyzed. The results of PageRank and Connected Components are especially relevant because of the wide performance differences they show, taking into account that they process the same input

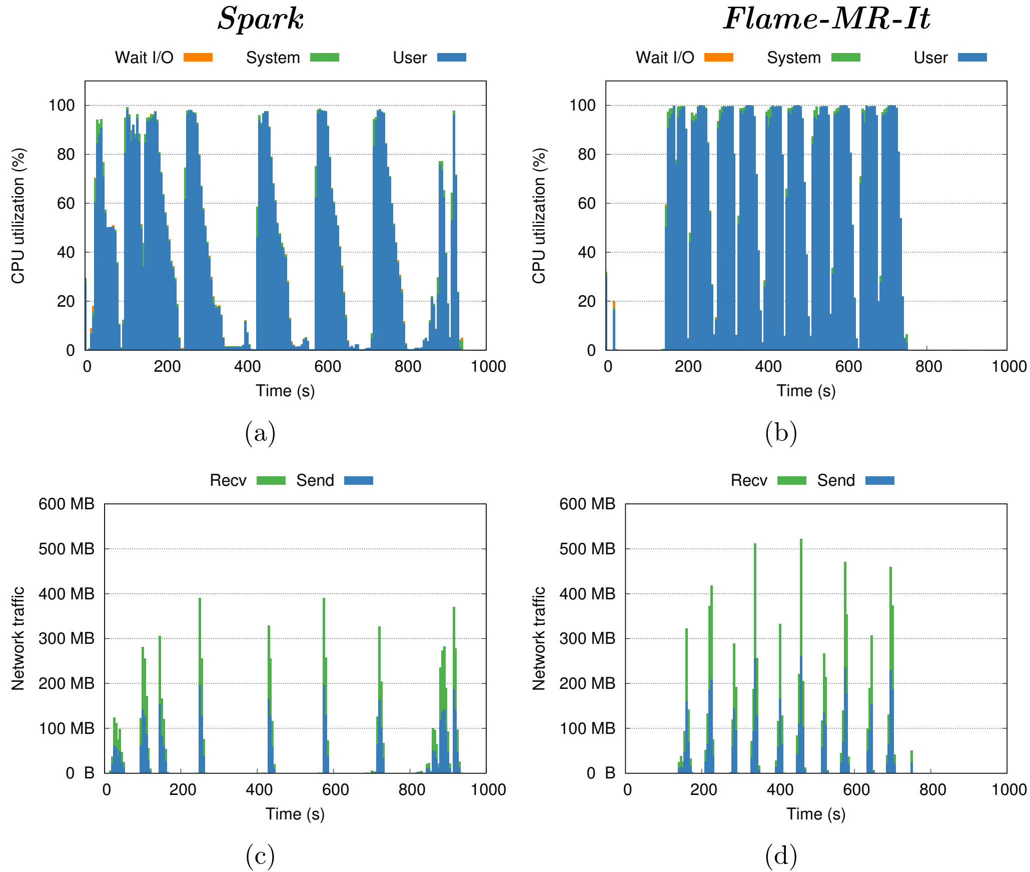


Figure 4.5: CPU utilization and network traffic for PageRank in i2.4xlarge

dataset using different graph algorithms. Figures 4.5 and 4.6 depict the CPU utilization and network traffic of PageRank and Connected Components, respectively, for i2.4xlarge instances. Although disk and memory utilization have also been analyzed, the disk traffic is almost negligible during the entire execution of both frameworks (below 14%). The main difference in the memory utilization is that Spark consumes 62% and 25% more memory than Flame-MR-It for PageRank and Connected Components, respectively. Apart from that, both frameworks show the same pattern that does not bring any meaningful insight. Therefore, disk and memory utilization graphs are not shown for clarity purposes.

The iterative behavior of both Spark and Flame-MR-It for PageRank can be clearly seen in Figure 4.5. The CPU utilization peaks correspond with the jobs performed during the workload (see Figures 4.5a and 4.5b). Network traffic also

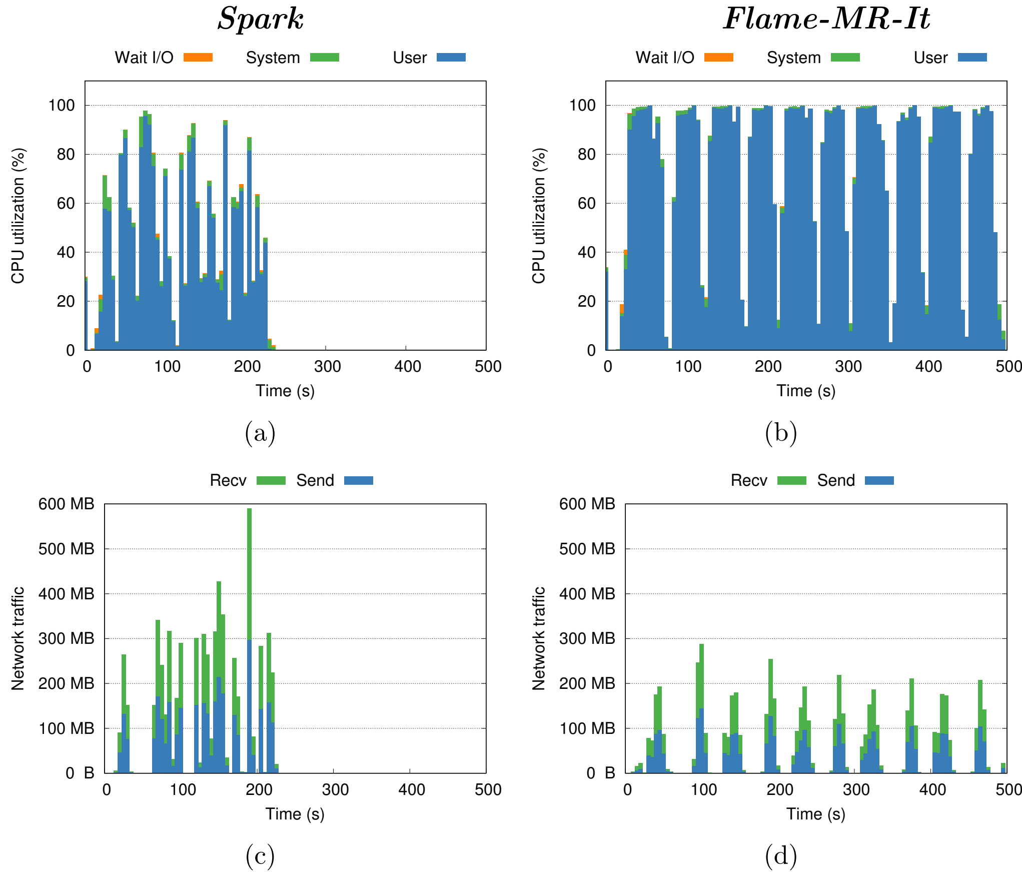


Figure 4.6: CPU utilization and network traffic for Connected Components in i2.4xlarge

shows similar peaks of activity that belong to the data shuffling within each job (Figures 4.5c and 4.5d). Note that Flame-MR-It shows a gap without activity at the beginning of the workload (Figure 4.5b), which is due to the data initialization performed by the MR Driver in PageRank. Regarding Connected Components, it can be seen that the iterative behavior of the workload is present in Flame-MR-It (Figures 4.6b and 4.6d) but is not so clear in Spark (Figures 4.6a and 4.6c). As previously commented, the optimized GraphX library leverages the Spark operations in a more efficient way than the MapReduce counterpart, thus benefiting from improved operation scheduling and in turn better performance. Another remark is that the best performer is the one with the highest network traffic: Flame-MR-It for PageRank (Figure 4.5d) and Spark for Connected Components (Figure 4.6c). Therefore, network bandwidth is not a performance bottleneck for any of these

workloads.

As a summary, the average performance improvement that Spark and Flame-MR-It obtain, compared to Hadoop, is 63% and 48%, respectively, when using i2.4xlarge instances. These results confirm the overall efficiency of our memory optimizations, as Flame-MR-It can transparently improve the performance of MapReduce workloads, while providing competitive results compared to Spark. Using one or another to improve the performance of an existing MapReduce application would depend on the particular workload characterization and the willingness of the user to modify the source code (if available).

4.3. Applicability study: optimization of real-world use cases

The previous section has evaluated the performance benefits provided by Flame-MR when executing standard benchmarks. These are synthetic workloads that do not always reflect the characteristics of the applications executed in real-world use cases. Hence, this section selects three MapReduce applications that are commonly used to process large-scale datasets, analyzing their MapReduce implementations and describing how Flame-MR adapts to the characteristics of each one. The performance comparison between Flame-MR[1] and Hadoop analyzes the benefits for each specific use case.

4.3.1. VELaSSCo: data visualization queries

VELaSSCo [74] is a Big Data visualization architecture that relies on the MapReduce model to extract information from simulation datasets. This section first provides more details of this project and describes the main characteristics of the MapReduce workloads. Next, the main challenges of running these workloads with Flame-MR are explained, analyzing its benefits by presenting the experimental configuration and performance results. Finally, some concluding remarks are provided.

[1] As Flame-MR-It has shown better performance and resource utilization in the previous section, this version is evaluated in this section relabeled as "Flame-MR".

Overview

VELaSSCo is a query-based visualization framework that aims at providing users with a tool to manipulate and visualize large simulation datasets. These datasets are generated by large parallel simulations relying on Finite Element Methods (FEM) or Discrete Element Methods (DEM). For both methods the simulation updates the properties of the nodes (FEM) or particles (DEM) at each time step. The user runs a 3D visualization client to request the execution of specific visualization algorithms on given parts of the data. The query is sent to the VELaSSCo cluster and translated into a Hadoop job that queries the input data and performs the expected data transformation. The result is sent back to the client for the final 3D rendering and display. The VELaSSCo architecture can be decomposed into three subsystems: client, analytics and storage, described next.

The client subsystem provides data visualization to the user, generating a new query when the user performs an action. Each query has a certain type depending on the action performed by the user. Analytical queries are the ones that require to extract some information from the dataset by means of a MapReduce workload (e.g. calculating the bounding box of a model). The analytics subsystem is in charge of receiving these queries and determining the computation needed to complete each one. That computation is performed by a MapReduce workload on a Hadoop cluster. The workloads employed in VELaSSCo consist of a single MapReduce job that typically operates over a subset of the dataset (a few simulation time steps for instance). Finally, the data persistence is performed by the storage subsystem. This subsystem employs HDFS to distribute the data among the computing nodes of the cluster. It also relies on HBase [9], a database system on top of HDFS, to allow the extraction of parts of the dataset without reading it entirely. This optimizes the amount of I/O operations needed to perform the computations. Instead of searching the relevant data through the entire dataset, the MapReduce workload uses the key-value format provided by HBase to fetch the required elements. The indexed system used in HBase accelerates the retrieving operation by avoiding the reading of unnecessary data.

As VELaSSCo is a real-time visualization platform, the performance of the actions executed by the user is crucial to ensure an appropriate user experience. Howe-

ver, Hadoop is not able to achieve this goal when dealing with large-scale datasets. This use case focuses on the acceleration of the queries used in the analytics subsystem of VELaSSCo by using Flame-MR.

MapReduce implementation

We list below the main analytical queries included in VELaSSCo:

- GetBoundingBoxOfAModel (BB): Computes the spatial bounding box for the selected dataset, i.e the min and max coordinate of the enclosed elements in the x, y and z dimensions.
- GetBoundaryOfAMesh (BM): Computes the set of elements that are at the boundary of the selected mesh, i.e. the surface given by the triangles belonging to only one mesh cell.
- GetListOfVerticesFromMesh (LVM): Obtains a list of identifiers (IDs) of the elements contained in a mesh.
- GetMissingIDsOfVerticesWithoutResults (MIV): Obtains the IDs of those mesh elements that do not contain any simulation result.
- GetSimplifiedMesh (SM): Obtains a simplified version of the mesh model, reducing the total dataset size by combining nearby elements.

As previously mentioned, these queries are performed by MapReduce workloads that are composed of a single job. All jobs extract the input data from HBase, selecting the relevant elements according to the information provided by the user. To read the data, the MapReduce implementation is based on a custom input formatter provided by HBase, which is used by the mappers to iterate over the entries allocated to them. Once the output of the job is calculated, it is converted to text files and stored in HDFS in order to be accessible by the client.

The implementation of each query includes the definition of the map and reduce functions. These functions use custom data types defined in VELaSSCo, which implement the Writable interface required for data serialization. So, map and reduce functions are configured to use these data types when reading and writing data.

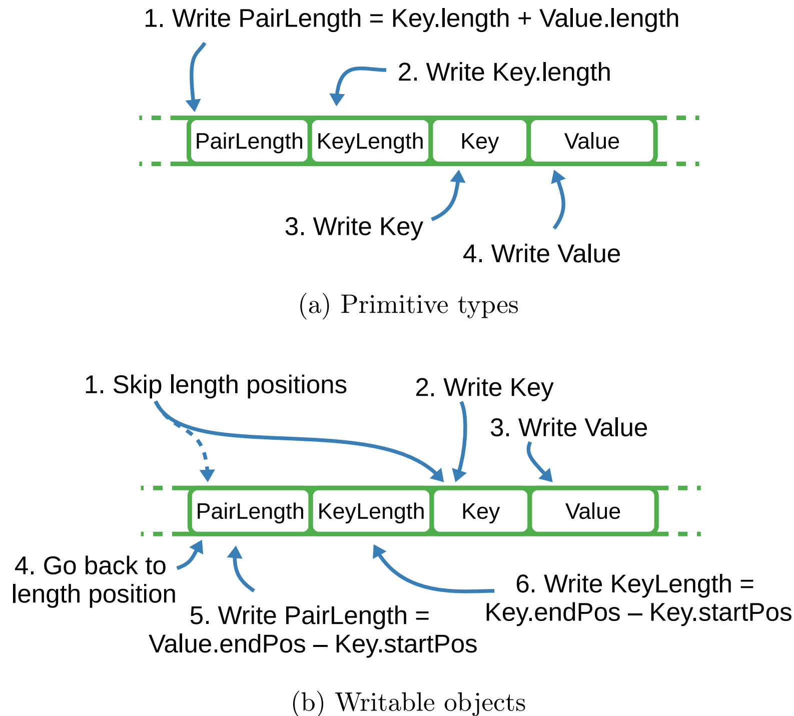
Challenges

The use of Flame-MR to optimize the VELaSSCo queries must take into account the characteristics that differ from standard Hadoop jobs. In particular, reading input data from HBase and using custom data types must be handled correctly to avoid incompatibility problems. This section describes how they are supported in Flame-MR.

Flame-MR is oriented to processing large textual datasets stored in HDFS, which is a common use case in MapReduce applications. Therefore, the reading of input data has been designed to make the common case fast. When launching a map operation, Flame-MR connects to HDFS and reads a full input split (e.g. 256 MB) to memory by copying the data to a set of medium-sized DataBuffers (e.g. 1 MB) allocated in the DataPool, as explained in Section 3.3.1. Once the input split is read, the connection to HDFS is closed and the DataBuffers are parsed in memory to obtain the input pairs and feed the mappers.

The in-memory parsing mechanism of Flame-MR is only possible when the input dataset is stored in HDFS in textual format. For other formats, the data source is unknown, and the software interface defined by Hadoop only allows reading the input pairs one by one. Therefore, copying an input split entirely to memory is not allowed and the reading of input data needs to be addressed differently. That is the case of VELaSSCo, which reads the data from HBase. When a map operation is launched in Flame-MR, the input formatter connects to the HBase server to read the data contained in the input split. Then, the map operation uses the interfaces provided by the input formatter class to read the input pairs, passing them to the user-defined map function. By doing this, the correct functioning of the queries is ensured. Note that this behavior can be extrapolated to any formatter class.

The use of custom data objects in VELaSSCo has also implications for Flame-MR. This happens because Flame-MR modifies the behavior of primitive Hadoop data types, like text and numerical types, in order to optimize read and write operations (see Section 3.3.1). These modifications include the use of in-memory addressing of serialized data to avoid the creation of data objects in sort and copy operations. When a key-value pair is stored in a DataBuffer, a header is added to indicate the pair and key lengths, which will be used to read data without creating



(a) Primitive types

(b) Writable objects

Figure 4.7: Data object serialization in Flame-MR

the objects. Therefore, the implementation of primitive data types in Flame-MR is extended with additional methods to obtain the length of data objects before writing them to the buffer. As VELaSSCo implements specific data objects inside each query, Flame-MR must adapt its behavior to comply with the standard Writable interface, which does not provide any information about the length of the objects.

The serialization mechanism of Hadoop primitive data types in Flame-MR is shown in Figure 4.7a. The pair and key length are calculated before writing the key-value pair to the buffer. To obtain the same results with custom Writable objects, Flame-MR performs the mechanism shown in Figure 4.7b. Pair and key lengths are unknown beforehand, so their positions must be skipped, writing the key-value pair after them. Once the data has been written, the lengths are calculated according to the writing position after copying the key-value pair. The lengths are then written by going backwards on the DataBuffer to the original position. This mechanism ensures compatibility with all types of Writable objects, while maintaining the in-memory optimizations of Flame-MR.

Experimental configuration

Next, we will describe the experimental testbed used in the comparison between Hadoop and Flame-MR when executing the VELaSSCo queries. The experiments have been conducted in the Grid'5000 infrastructure [50]. Two cluster sizes (n) have been used: 17 and 25 nodes with 1 master and n-1 slaves. These nodes are equipped with 2 Intel Haswell-based processors with 8 physical cores each (i.e. 16 cores per node), 128 GB of memory and 2 local disks of 558 GB each (see Table 4.4 for more details). The experiments have used HBase 1.2.4, Hadoop 2.7.3 and Flame-MR 1.1. The configuration of the frameworks has been carefully set up by following their user guides, taking into account the characteristics of the systems (e.g. number of CPU cores, memory size). The most important parameters of the resulting configuration are shown in Table 4.5.

The VELaSSCo queries used in the evaluation are the ones described previously. The input dataset has been extracted from a FEM simulation that represents the wind flow in the city of Barcelona with an eight-meter resolution. This dataset has 12,089,137 vertices and occupies 367 GB. For each query, the median elapsed time of 10 executions was calculated, although the standard deviations observed were not significant.

Performance results

Figures 4.8a and 4.8b show the execution times of VELaSSCo queries using 17 and 25 nodes, respectively. Flame-MR widely outperforms Hadoop with both cluster sizes, showing an average reduction in execution time of 87% with 17 nodes and 88% with 25 nodes. This reduction is due to the more efficient architecture of Flame-MR, which can better leverage the memory and CPU resources of the system. Note that each Worker process in Flame-MR can schedule multiple map and reduce operations, allocating them to the cores available as they become idle. Therefore, the Worker can use the same HBase connection for all map operations. Instead, Hadoop allocates a single Java process to each map and reduce task, and so it creates an HBase connection for each one, increasing the overhead. This enables Flame-MR to process more HBase requests per time unit compared to Hadoop, which is reflected in the information counters provided by HBase.

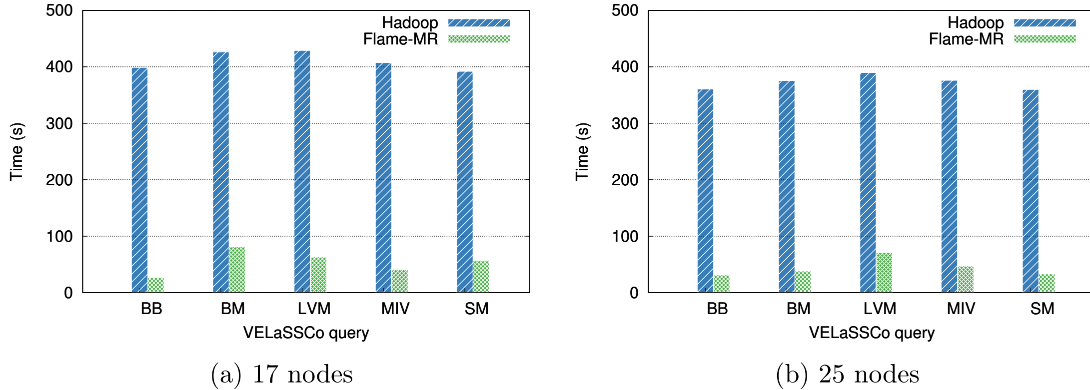
Table 4.4: Node characteristics of Grid'5000

Hardware configuration	
CPU model	2 × Intel Xeon E5-2630 v3 (Haswell)
CPU speed (Turbo)	2.40 GHz (3.20 GHz)
#Cores	16
Cache (L1 / L2 / L3)	32 KB / 256 KB / 20 MB
Memory	128 GB DDR4 2133 MHz
Disk	2 × 558 GB HDD
Network	4 × 10 Gbps Ethernet
Software configuration	
OS version	Debian Jessie 8.5
Kernel	3.16.0-4
Java	Oracle JDK 1.8.0_121

Table 4.5: Configuration of the frameworks in Grid'5000

Hadoop		**HDFS**	
Mapper/Reducer heap size	7.25 GB	HDFS block size	256 MB
Mappers per node	8	Replication factor	3
Reducers per node	8		
Shuffle parallel copies	20		
IO sort MB	1600 MB		
IO sort spill percent	80%		

Flame-MR	
Workers per node	2
ThreadPool size	8
Worker heap size	44 GB
DataPool size	30.8 GB
DataBuffer size	1 MB



(a) 17 nodes

(b) 25 nodes

Figure 4.8: Execution times of VELaSSCo queries with Hadoop and Flame-MR

Remarks

This section addressed the optimization of analytical queries used to process datasets stored in HBase. These queries implement custom input formats and data types by using the class interfaces provided by Hadoop. Flame-MR is able to adapt to these characteristics without hindering the optimizations implemented in its underlying in-memory architecture. Using Flame-MR, the performance of the queries is improved by almost one order of magnitude, enhancing the user experience of VELaSSCo.

4.3.2. CloudRS: error removal in genomic data

This section addresses the optimization of CloudRS [21], a bioinformatics tool that detects and corrects errors in large genomic datasets. First, an overview of the purpose of CloudRS is presented and the main characteristics of its MapReduce implementation are provided. Next, the challenges of using Flame-MR to optimize its performance are introduced, presenting also the experimental configuration and performance results. Finally, the last part of the section extracts some conclusions from this practical use case.

Overview

The datasets generated by Next Generation Sequencing (NGS) platforms are composed of a large number of DNA sequence fragments, which are small pieces of genomic information contained in a string of characters (called reads). Each character of a read represents a DNA base, namely Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). The analysis of these datasets is performed by processing the sequences and identifying relationships between them.

During the generation of genomic datasets, NGS sequencers often introduce errors by placing incorrect bases in the reads. This can affect the quality of the results obtained by downstream analysis, and so it is usually minimized by introducing an error correction phase in the preprocessing stage of the NGS pipeline. In fact, this is a critical step in NGS workflows like *de novo* genome assembly or DNA resequencing. CloudRS is a popular tool for performing this preprocessing task, being based on the ReadStack (RS) algorithm [47]. This algorithm makes use of the characteristics of NGS datasets to identify common patterns in the sequences and correct the mismatching ones.

The DNA sequences that compose a dataset are not necessarily disjoint, as they can share information due to the overlap of reads performed by the sequencer. CloudRS takes advantage of this characteristic to identify redundant information in the sequences and correct errors in the bases. First, it splits each sequence into several subsequences. Second, it compares the different candidates for each subsequence, choosing the one that appears most of the times.

CloudRS is implemented with the MapReduce model by operating over datasets stored in HDFS. As it is a common step in large NGS workflows, its performance is crucial to obtain the results of the analysis in a reasonable time. For that reason, it has been chosen to be optimized with Flame-MR. Further details about its implementation are provided in the next section.

MapReduce implementation

CloudRS is an iterative workload that follows several phases to process the input dataset, explained below:

1. LoadReads: This phase prepares the input dataset to be processed, discarding noisy information and converting the sequences into a more suitable format for Hadoop. In order to avoid the comparison of very repetitive sequences, it also builds a list of the most frequent subsequences. Later, this list is used to filter them out and avoid workload imbalance.

2. PreCorrection: Each sequence is split into different subsequences that are candidates in the next phases. The candidates for each subsequence are aligned to allow their comparison, using a wildcard pattern.

3. ErrorCorrection: The set of subsequence candidates is iterated through by using the information obtained in the previous phases. First, the most frequent subsequences are filtered out. Then, the candidates are compared by emitting a vote for each position. When all the votes have been emitted, the correct alternative is chosen by majority. This calculation repeats several times until the obtained subsequences remain invariant.

4. Screening: Once the correct subsequences have been calculated, the input dataset is reprocessed to fix the errors, replacing each subsequence with its corresponding correct alternative.

5. Conversion: The output dataset is converted to a standard format in order to be processed by subsequent NGS applications (e.g. sequence alignment).

Using these five phases, the execution of CloudRS involves a total of 11 MapReduce jobs, some of them being repeated in the ErrorCorrection phase. Their implementation uses an old version of the Hadoop API, although this only affects the interfaces used by the source code of the workload. CloudRS also makes use of the DistributedCache feature provided by Hadoop to make the list of most frequent sequences available to the mappers during the ErrorCorrection phase.

The input and output formatter classes in CloudRS are standard ones that operate over textual data stored in HDFS. Instead of using a custom formatter, CloudRS gives format to the data within the user-defined map and reduce functions. CloudRS uses standard Hadoop Text objects to represent the data as strings, separating the different fields by using special characters. Note that this is a very inefficient implementation compared to the use of a custom formatter that can represent in-memory

data as binary objects. The approach of CloudRS requires to parse data objects from textual data, while also having to convert them to strings when writing the output.

Challenges

As explained previously, CloudRS is an iterative workload that executes several MapReduce jobs to obtain the final result. The resource management of Flame-MR adapts better to this kind of computation than Hadoop, thus providing better performance. On the one hand, Hadoop allocates one Java process per map/reduce task. Therefore, Hadoop needs to create many map and reduce processes at the start of each job, stopping them when the job is finished. On the other hand, Flame-MR deploys a single Java process per Worker and uses a ThreadPool to execute the map and reduce functions, as described in Section 3.3.1. These processes are reutilized between MapReduce jobs until the entire workload is finished. Note that Flame-MR also benefits from the reutilization of internal data structures like the allocation of memory buffers.

Regarding data input and output, Flame-MR is oriented to the processing of large textual datasets, as previously mentioned. Therefore, it uses optimized input and output formatters that minimize the amount of connections to HDFS. Similarly, the implementation of textual data objects used in Flame-MR reduces the amount of memory copies and object creations when performing sort and copy operations. CloudRS makes use of both characteristics, and so it is especially well suited to be optimized with Flame-MR. However, the inefficient data formatting explained before is intrinsic to CloudRS, as it is performed inside the map/reduce functions. The goal of Flame-MR is to improve applications' performance without modifying their source code, and so we cannot modify those user-defined functions. Therefore, the inefficiency of the data formatting will also be present in Flame-MR, although it is alleviated by using its efficient implementations of textual data objects.

The use of the old Hadoop API is also supported in Flame-MR by connecting old classes and methods with its corresponding counterparts in the new API. Furthermore, Flame-MR supports the use of the DistributedCache by copying the data files required by the mappers to the computing nodes where they are being executed,

thus making the data available to the application.

Experimental configuration

The experimental configuration used to evaluate CloudRS with Hadoop and Flame-MR is described below. As genomic applications are executed in many kinds of systems, the evaluation has considered two different scenarios: the use of a private cluster with 9 nodes, Pluton, and a public cloud platform, Microsoft Azure [88], with 17 and 25 instances. As in the case of VELaSSCo, each cluster size n corresponds to 1 master and n-1 slaves.

The hardware and software characteristics of Pluton and Azure are shown in Tables 4.6 and 4.7, respectively. Pluton nodes are equipped with 16 cores each, 64 GB of memory and one local disk of 1 TB, being interconnected via InfiniBand FDR and Gigabit Ethernet. Azure experiments have been carried out using L16S instances located in the West Europe region. These instances have 16 cores per node, 128 GB of memory and a local SSD disk of 2.7 TB. The experiments have used Hadoop 2.7.4 and Flame-MR 1.1. The configuration has been adapted to the characteristics of the systems, resulting in the parameters shown in Tables 4.8 and 4.9 for Pluton and Azure, respectively. Furthermore, both frameworks have used the IPoIB interface available in Pluton, which allows taking advantage of the InfiniBand network via the IP protocol. Finally, some parameters such as the HDFS block size have been experimentally tuned to obtain the best performance on each system.

The input dataset used in the experiments is SRR921890, which has been obtained from the DDBJ Sequence Read Archive (DRA) [28]. It is composed of 16 million sequences of 100 bases each (5.2 GB in total). The results shown correspond to the median elapsed time of 10 executions. The standard deviations observed were not significant. In this use case, the experiments have been conducted by using BDEv.

Table 4.6: Node characteristics of Pluton

Hardware configuration	
CPU model	2 × Intel Xeon E5-2660 (Sandy Bridge)
CPU speed (Turbo)	2.20 GHz (3 GHz)
#Cores	16
Cache (L1 / L2 / L3)	32 KB / 256 KB / 20 MB
Memory	64 GB DDR3 1600 MHz
Disk	1 TB HDD
Networks	InfiniBand FDR & GbE
Software configuration	
OS version	CentOS release 6.8
Kernel	2.6.32-642
Java	Oracle JDK 1.8.0_45

Table 4.7: Node characteristics of L16S instances in Azure

Hardware configuration	
CPU model	Intel Xeon E5-2698B v3 (Haswell)
CPU speed	2 GHz
#Cores	16
Cache (L1 / L2 / L3)	32 KB / 256 KB / 40 MB
Memory	128 GB
Disk	2.7 TB SSD
Network	4 × 10 Gbps Ethernet
Software configuration	
OS version	CentOS release 7.4
Kernel	3.10.0-514
Java	OpenJDK 1.8.0_151

Table 4.8: Configuration of the frameworks in Pluton

Hadoop		HDFS	
Mapper/Reducer heap size	3.4 GB	HDFS block size	256 MB
Mappers per node	8	Replication factor	3
Reducers per node	8		
Shuffle parallel copies	20		
IO sort MB	841 MB		
IO sort spill percent	80%		

Flame-MR	
Workers per node	4
ThreadPool size	4
Worker heap size	11.8 GB
DataPool size	8.3 GB
DataBuffer size	512 KB

Table 4.9: Configuration of the frameworks in Azure

Hadoop		HDFS	
Mapper/Reducer heap size	6.7 GB	HDFS block size	128 MB
Mappers per node	8	Replication factor	3
Reducers per node	8		
Shuffle parallel copies	20		
IO sort MB	1704 MB		
IO sort spill percent	80%		

Flame-MR	
Workers per node	4
ThreadPool size	4
Worker heap size	24 GB
DataPool size	16.8 GB
DataBuffer size	512 KB

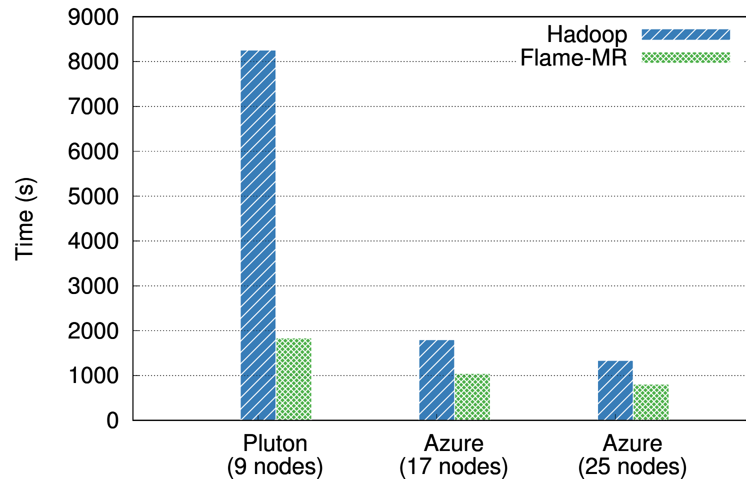


Figure 4.9: Execution times of CloudRS with Hadoop and Flame-MR

Performance results

Figure 4.9 shows the performance results of CloudRS. As can be seen, Flame-MR clearly outperforms Hadoop in both testbeds. In fact, Flame-MR obtains a 78% reduction in execution time in Pluton. In the case of Azure, it obtains a reduction of approximately 40% for both cluster sizes. Note that Hadoop presents a huge performance improvement when scaling from Pluton with 9 nodes to Azure with 17 nodes. In addition to the double amount of slave nodes, this improvement is due to the better node characteristics of Azure. Compared to Pluton, Azure provides a more recent CPU microarchitecture and doubles the available memory, while the SSD disk decreases I/O waiting times. Furthermore, the execution time of Flame-MR with 9 nodes in Pluton is almost the same as using Hadoop in Azure with 17 nodes. Therefore, Flame-MR allows reducing the execution time of CloudRS without needing to increase the computational resources, obtaining a performance improvement equivalent to using a double-sized cluster in this specific case, minimizing incurred costs in public cloud platforms such as Azure.

Remarks

The use of Flame-MR to optimize CloudRS has shown an important reduction in execution time. Taking into account that the data formatting inefficiency of the

source code of the application cannot be avoided, this use case is a good example of how Flame-MR can reduce the performance impact of those inefficiencies, without redesigning the software or employing further computing resources.

4.3.3. MarDRe: duplicate read removal in genome sequencing data

This section addresses the optimization of MarDRe [35], a bioinformatics application that removes duplicate reads in large genomic datasets. The first part of the section describes the main functionalities and characteristics of MarDRe, providing the details of its MapReduce implementation. After the challenges of optimizing MarDRe with Flame-MR are explained, the experimental configuration and results show the performance benefits of Flame-MR. Finally, some conclusions are extracted from this use case.

Overview

As explained in Section 4.3.2, genomic datasets generated by NGS sequencers contain redundant information due to the existence of overlapped reads. This characteristic causes the appearance of duplicate or near-duplicate sequences in large datasets, which neither provide new information nor improve the results of analytical processes. However, processing them consumes system resources and wastes execution time. Therefore, they are often removed to decrease the overall runtime of the downstream analysis.

MarDRe is a MapReduce application that is used to detect and remove these duplicate sequences in genomic datasets stored in HDFS. It is based on a clustering mechanism that groups the sequences by similarity. Then, the sequences within a group are compared by using an optimized algorithm that discards the sequences that do not provide new information.

As in the case of CloudRS, MarDRe is usually performed in the preprocessing stage. Therefore, reducing its execution time can have a significant impact on the performance of the overall NGS pipeline. Further details of its implementation are

provided below.

MapReduce implementation

The MapReduce workload used in MarDRe performs a single Hadoop job to process the data stored in HDFS. In contrast to CloudRS, MarDRe supports input datasets stored in FASTQ/FASTA, which are standard formats commonly employed in genomic datasets.

The map phase is used to cluster the DNA sequences into groups. Each mapper reads the data belonging to its input split by using a custom formatter that reads the sequences in FASTQ/FASTA format. The input sequences are then divided into prefix and suffix to group the ones that share the same prefix. During the shuffle phase, the prefix is used as key to partition and sort the map output pairs. The value of the pair contains the sequence information by using a custom data type defined in MarDRe. Next, the map output pairs are sent to the reducer nodes where they are processed. Once the reducers receive all the assigned sequences, they carry out the comparison to filter out the duplicates by using the optimized algorithm presented in [49]. This algorithm does not compare all the sequences within each group, but uses the first one as a reference for the rest. If the number of mismatches of a sequence with respect to the first one is higher than a user-defined threshold, the sequence is discarded. Moreover, the bases of the sequences are not compared one by one. Instead, a 4-bit encoding is used to represent the bases, determining the differences by using a bit-wise XOR operation. After that, the output of the reducers containing the remaining sequences without duplicates is written to HDFS in FASTQ/FASTA format.

MarDRe is especially well suited to the MapReduce model, as the main part of its clustering algorithm is performed by the underlying grouping-by-key mechanism of Hadoop. Furthermore, its implementation leverages the use of custom formatters and data objects to avoid inefficient parsing of the input dataset. Although MarDRe shows good performance with balanced workloads, real-world datasets are highly skewed, with lots of sequences that share a common prefix. This situation introduces important load balancing problems in the reduce phase due to the comparison of large sets of sequences. In turn, this causes some reducers to have

excessive execution times. As a MapReduce job has to wait for all reducers to finish, the load balance problem in the reduce phase affects the overall performance. The next section discusses how Flame-MR solves this problem in a transparent way.

Challenges

Flame-MR must adapt to the characteristics of MarDRe when optimizing its performance. As in the VELaSSCo use case, the use of custom formatters and data objects in MarDRe requires the utilization of the standard API provided by Hadoop, while keeping the in-memory optimizations.

Regarding the load balancing problem explained before, the standard behavior of Flame-MR emulates the way Hadoop processes the data without modifying the operation of the map and reduce functions. Therefore, load imbalance also affects Flame-MR. To alleviate it without changing the source code of the application, a new load balancing mode has been developed in Flame-MR version 1.1. During the reduce phase, large partitions are detected and split into several chunks. In doing so, the computation is parallelized and the execution times of heavy-loaded reducers are decreased.

Figure 4.10 illustrates the operation of the load balancing mode. Instead of processing a large partition with a single reduce operation, the data is split into different chunks with a maximum size calculated upon the number of chunks defined by the user. Next, each chunk is reduced in parallel, writing the output to HDFS. Note that this mechanism is likely to introduce changes in the output results of the reduce phase, as the input pairs are passed to the reduce function in different groups. Therefore, the load balancing mode is only applicable to those Hadoop jobs that can support modifications in the reduce partitioning without affecting the logic of the application, even if the final output suffers slight variations. In the particular case of MarDRe, the splitting of partitions leads to different comparisons to be done between sequences. Although this may modify the actual sequences that are filtered in the end, it does not affect the purpose of the workload as long as the percentage of sequences filtered does not vary significantly. For example, in the experimental results provided next, the amount of duplicate reads filtered did not vary more than 0.02% when using the load balancing mode.

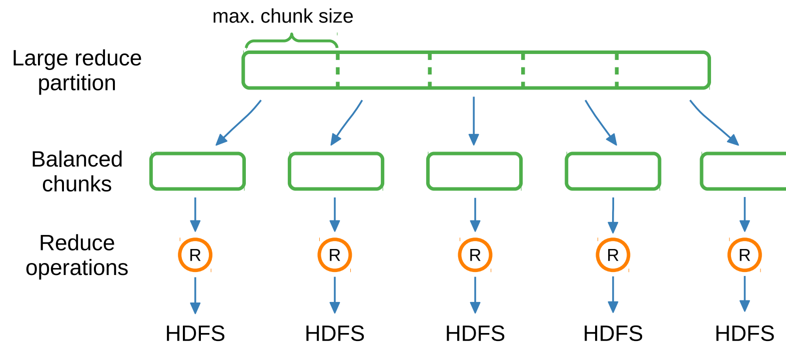


Figure 4.10: Load balancing mode in Flame-MR version 1.1

Experimental configuration

MarDRe and CloudRS are both executed as preprocessing steps of an NGS analysis on Big Data infrastructures. Therefore, the evaluation of MarDRe has employed the same experimental configuration as CloudRS using Pluton and Azure as testbeds (see Tables 4.6, 4.7, 4.8 and 4.9 in Section 4.3.2), and the BDEv tool to conduct the evaluation. However, the computational requirements of MarDRe are significantly lower than CloudRS, and so a larger dataset was used in these experiments: SRR377645. This dataset is composed of 214 million reads of 100 bases each (67 GB).

The evaluation includes the results of Hadoop, Flame-MR and Flame-MR with the load balancing mode activated (labeled as Flame-MR-LB in the graphs). By default, the number of chunks used in the load balancing mode is set to the number of cores of the Worker. In the experiments, this value has been tuned for improved performance, splitting each partition in 13 and 9 chunks in Pluton and Azure, respectively.

Performance results

Figure 4.11 shows the execution times of MarDRe with Hadoop, Flame-MR and Flame-MR-LB. As can be seen, Flame-MR outperforms Hadoop by 43% in Pluton and by approximately 24% in Azure for both cluster sizes. The improvement provided by Flame-MR-LB is even better, reducing the execution time of Hadoop by 66% both in Pluton and Azure (17 nodes), and by 77% when using 25 nodes

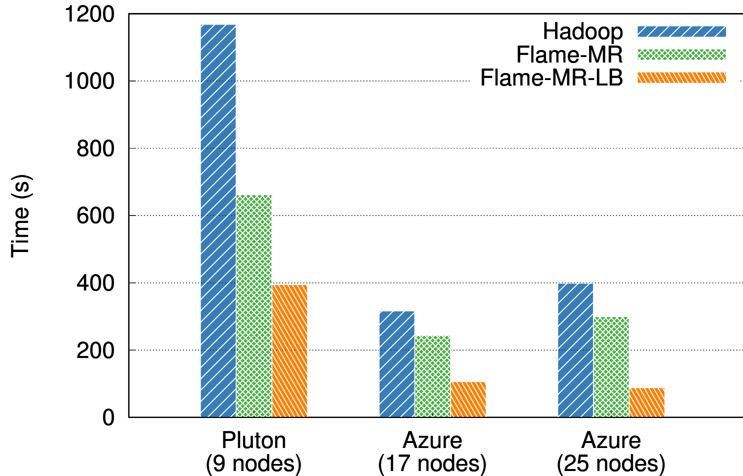


Figure 4.11: Execution times of MarDRe with Hadoop, Flame-MR and Flame-MR-LB

in Azure. This huge improvement demonstrates the effectiveness of the load balancing mode explained previously, together with the efficient in-memory architecture of Flame-MR.

Note that the execution times of Hadoop and Flame-MR using 25 nodes are higher than with 17 nodes, which is due to the workload imbalance problem. With more nodes and thus more reducers, the load per reducer is decreased, but the reducers that process the largest partitions require the same time. This issue, together with the additional overhead of managing more nodes, hinders the performance of both frameworks. However, Flame-MR-LB does not present this problem, obtaining slightly better results with 25 nodes than with 17.

In order to provide more information about the load balance problem of MarDRe, Table 4.10 shows the processing time of the fastest, median and slowest reducer compared to the overall execution time. As can be seen, the time consumed by the slowest reducer is clearly correlated with the overall execution time of the application. Furthermore, there exist huge differences between the fastest and slowest reducers. In the case of Hadoop and Flame-MR, the use of larger clusters decreases the processing time of the fastest and median reducers. This does not always happen with the slowest reducer, which consumes more time with 25 nodes than with 17 for both frameworks. This fact, along with the overhead of managing more nodes, causes

Table 4.10: Load balancing in MarDRe

(a) Pluton (9 nodes)

	Reducer time			
	Fastest	**Median**	**Slowest**	**Execution time**
Hadoop	25.7s	136s	972.2s	1168.3s
Flame-MR	14.9s	19.7s	435.7s	662.5s
Flame-MR-LB	0.004s	1.1s	184.2s	394.7s

(b) Azure (17 nodes)

	Reducer time			
	Fastest	**Median**	**Slowest**	**Execution time**
Hadoop	12.7s	22.1s	259.4s	316.4s
Flame-MR	11.8s	15.8s	203.1s	243.9s
Flame-MR-LB	0.001s	1.1s	60.5s	106.3s

(c) Azure (25 nodes)

	Reducer time			
	Fastest	**Median**	**Slowest**	**Execution time**
Hadoop	8.9s	13.8s	346.3s	398.7s
Flame-MR	6s	8.2s	254.9s	299.5s
Flame-MR-LB	0.001s	0.8s	29.1s	88.1s

the overall execution time to be higher. Flame-MR-LB shows a different behavior. When using larger clusters, the fastest and median reducers remain almost invariant, while the slowest one consumes less time. This, in turn, reduces the overall execution time of Flame-MR-LB.

Remarks

This section has shown the benefits of optimizing MarDRe with Flame-MR. Without modifying its source code, we obtain significant performance improvements

by better leveraging the system resources. Furthermore, the new load balancing mode available in Flame-MR has demonstrated its usefulness to reduce the impact of skewed loads in the reduce phase, reducing up to 77% the execution time of Hadoop.

4.4. Conclusions

This chapter has addressed the evaluation of Flame-MR using several use cases and systems. First, Flame-MR has been compared with Hadoop on different Amazon EC2 instances, obtaining lower execution times for all benchmarks, with a maximum reduction of 65%. These performance differences become wider when using larger memory sizes. Therefore, Flame-MR is able to provide better in-memory computing capabilities than Hadoop, making it more suited for future computing systems with increasingly larger memory. Compared to representative in-memory computing frameworks like Spark, Flame-MR provides very competitive performance but without modifying the source code of the applications.

Second, the performance benefits of Flame-MR in real-world use cases have been assessed by using three different workloads from two application domains: visualization queries (VELaSSCo) and preprocessing of genomic datasets (CloudRS and MarDRe). On the one hand, Flame-MR improves the execution time of the analytical queries of VELaSSCo by adapting its behavior to the custom input formats and data objects defined in the workload. On the other hand, the iterative algorithm performed by CloudRS is also accelerated, overcoming some of the inefficiencies of its underlying implementation. Finally, the use of Flame-MR in MarDRe has not only optimized the underlying Hadoop data engine but also alleviated its load balancing problems.

The execution of several standard benchmarks with distinct characteristics, together with the assessment of real-world applications on different cluster and cloud systems, has proved the significant performance benefits provided by Flame-MR over Hadoop.

Chapter 5

Conclusions and future work

Next, we summarize the main contributions of the Thesis and provide some insights on future research directions.

Conclusions

Over the last several years, the datasets managed by Big Data systems have been experiencing a steady increase in size and complexity, demanding higher data processing capabilities from distributed frameworks such as Apache Hadoop. As a consequence, analyzing and optimizing the performance of these frameworks has gained huge attention. This Thesis has focused on these issues by providing developers and users with new tools that, on the one hand, ease the task of evaluating Big Data frameworks and, on the other, accelerate existing Hadoop applications in a transparent way.

The assessment of Big Data frameworks has been addressed by developing the Big Data Evaluator (BDEv) tool, which eases the burden of setting up experimental testbeds using different frameworks and workloads. BDEv helps the user to unify the configuration of the different frameworks, facilitating a fair comparison among them. BDEv automatically configures the frameworks, launches their daemons over the cluster, generates the input datasets required for the experiments and executes the corresponding workloads. During their execution, it extracts the evaluation

metrics indicated by the user. These metrics are not only limited to performance and scalability, as BDEv also monitors resource utilization, energy efficiency and microarchitectural behavior. Several practical use cases have demonstrated the usefulness of BDEv to extract valuable information about the behavior of popular Big Data frameworks such as Hadoop, Spark and Flink.

Regarding the optimization of data processing pipelines, this Thesis has proposed Flame-MR, a MapReduce framework that accelerates existing Hadoop applications without any source code modification. It is based on an event-driven architecture that efficiently parallelizes data processing operations and movements, leveraging computational resources such as CPU cores and memory. The overall design of Flame-MR reduces the amount of redundant memory copies performed by Hadoop, while also using efficient sort and merge algorithms. Further optimizations of Flame-MR reutilize in-memory buffers to avoid unnecessary allocations and deallocations. Moreover, the caching of intermediate results has also reduced the overhead of disk and network operations in iterative workloads.

The performance benefits of Flame-MR have been thoroughly evaluated in different scenarios and systems. The use of standard benchmarks makes it easier to compare the performance results between frameworks. Using these benchmarks, Flame-MR reduces the execution time of Hadoop by 48% on average, while showing very competitive results compared to Spark. Although Spark obtains better performance for some iterative benchmarks, it is outperformed by Flame-MR when executing I/O-bound workloads like Sort and PageRank, with the additional benefit that it does not require the rewriting of existing Hadoop applications as with Spark.

Flame-MR has also been evaluated by using several real-world applications, including analytical database queries (VELaSSCo), error correction in genomic datasets (CloudRS) and duplicate read removal in genomic datasets (MarDRe). These applications, originally written for Hadoop, show an acceleration between 40% and 90%, which means that their execution time is reduced up to an order of magnitude in some cases. These results, along with the ones obtained using standard benchmarks, clearly demonstrate the performance benefits of Flame-MR over Hadoop.

Future work

The proposal of new evaluation and optimization tools for Big Data workloads sets the basis for future research lines that can benefit from the work conducted in this Thesis. As Big Data ecosystems are becoming more heterogeneous both in terms of hardware and software, there is need for intensified efforts to find new evaluation techniques to enable accurate comparisons of systems and paradigms.

Although execution time is generally used as the main performance metric for batch workloads, it cannot be applied to all scenarios (e.g. stream processing), which hinders performance comparisons between different paradigms. Therefore, there is need for new standard metrics, analogous to the ones utilized in more traditional parallel paradigms (e.g. Floating point Operations Per Second, FLOPS). These metrics must ease the extrapolation of results when varying experiment parameters like input data size or system characteristics like cluster size. Measuring the amount of data processed per unit time (MB/s) can be an initial approximation to such metric, but it is not always adequate, as the input data size can vary between scenarios due to format specifications. Moreover, iterative workloads do not spend all the execution time processing the input data. These issues must be carefully taken into account in order to elaborate a sound proposal for standard performance metrics in Big Data.

As frameworks evolve over time, they are becoming more complex to deploy, use and configure. This situation demands new evaluation tools and methods that cooperate with the frameworks in an active way. In addition to extracting evaluation metrics like resource utilization or energy efficiency, they must be able to use that information to improve performance. By analyzing the recorded data, the evaluation process must be fed back to adjust the configuration of the frameworks or adapt the system resources to the needs of the applications in a real-time fashion.

Data processing optimization is currently oriented to the performance improvement of a specific framework, solving existing issues in its design. However, the applicability of this approach is clearly limited to that specific framework. New optimizations should aim to find solutions to common performance problems, which may arise in multiple frameworks. Examples of that are data serialization or task scheduling issues. Finding efficient design solutions for them could provide deve-

lopers with valuable guidelines when designing new data processing frameworks or optimizing the existing ones.

The programming APIs currently exposed by Big Data processing frameworks often share many similarities among them. For example, Spark and Flink have a lot of data operators in common, such as *map()* or *filter()*. Therefore, the definition of a standard data processing API would be very helpful to homogenize the way operations are defined by these frameworks. Similarly to the way the MPI standard unified parallel programming in HPC, a common Big Data programming model would allow multiple implementations of the same operations, focusing on providing an efficient implementation for them while keeping compatibility between frameworks.

Bibliography

[1] Amazon Web Services Inc. Amazon Elastic Compute Cloud (Amazon EC2). https://aws.amazon.com/ec2/. [Last visited: October 2018].

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for Big Data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[3] AMPLab: a data warehouse benchmark. https://amplab.cs.berkeley.edu/benchmark/. [Last visited: October 2018].

[4] Apache Apex: enterprise-grade unified stream and batch processing engine. https://apex.apache.org/. [Last visited: October 2018].

[5] Apache Aurora: Mesos framework for long-running services and cron jobs. http://aurora.apache.org/. [Last visited: October 2018].

[6] Apache Cassandra: distributed NoSQL database. http://cassandra.apache.org/. [Last visited: October 2018].

[7] Apache Flink: scalable batch and stream data processing. http://flink.apache.org/. [Last visited: October 2018].

[8] Apache Hadoop. http://hadoop.apache.org/. [Last visited: October 2018].

[9] Apache HBase: Hadoop distributed Big Data store. https://hbase.apache.org/. [Last visited: October 2018].

[10] Apache Mahout: scalable machine learning and data mining. `http://mahout.apache.org/`. [Last visited: October 2018].

[11] Apache Pig: high-level language for data analysis. `https://pig.apache.org/`. [Last visited: October 2018].

[12] C. Avery. Giraph: large-scale graph processing infrastructure on Hadoop. In *2011 Hadoop Summit*, pages 5–9. Santa Clara, CA, USA, 2011.

[13] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguadé. Performance characterization of in-memory data analytics on a modern cloud server. In *5th IEEE International Conference on Big Data and Cloud Computing (BDCloud 2015)*, pages 1–8. Dalian, China, 2015.

[14] E. Baccarelli, N. Cordeschi, A. Mei, M. Panella, M. Shojafar, and J. Stefa. Energy-efficient dynamic traffic offloading and reconfiguration of networked data centers for Big Data stream mobile computing: review, challenges, and a case study. *IEEE Network*, 30(2):54–61, 2016.

[15] S. Bergamaschi, L. Gagliardelli, G. Simonini, and S. Zhu. BigBench workload executed by using Apache Flink. *Procedia Manufacturing*, 11:695–702, 2017.

[16] M. Bertoni, S. Ceri, A. Kaitoua, and P. Pinoli. Evaluating cloud frameworks on genomic applications. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015)*, pages 193–202. Santa Clara, CA, USA, 2015.

[17] C. Boden, A. Spina, T. Rabl, and V. Markl. Benchmarking data flow systems for scalable machine learning. In *4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR'17)*, pages 5:1–5:10. Chicago, IL, USA, 2017.

[18] R. Brun and F. Rademakers. ROOT – an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1-2):81–86, 1997.

[19] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1):285–296, 2010.

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 519–538. San Diego, CA, USA, 2005.

[21] C.-C. Chen, Y.-J. Chang, W.-C. Chung, D.-T. Lee, and J.-M. Ho. CloudRS: an error correction algorithm of high-throughput sequencing data based on scalable framework. In *2013 IEEE International Conference on Big Data (IEEE BigData 2013)*, pages 717–722. Santa Clara, CA, USA, 2013.

[22] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in Big Data systems: a cross-industry study of MapReduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[23] D. Cheng, P. Lama, C. Jiang, and X. Zhou. Towards energy efficiency in heterogeneous Hadoop clusters by adaptive task assignment. In *35th IEEE International Conference on Distributed Computing Systems (ICDCS'15)*, pages 359–368. Columbus, OH, USA, 2015.

[24] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *1st IEEE Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM'16)*, pages 1789–1792. Chicago, IL, USA, 2016.

[25] I. S. Choi, W. Yang, and Y.-S. Kee. Early experience with optimizing I/O performance using high-performance SSDs for in-memory cluster computing. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015)*, pages 1073–1083. Santa Clara, CA, USA, 2015.

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 143–154. Indianapolis, IN, USA, 2010.

[27] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: memory power estimation and capping. In *2010 International Symposium on*

Low-Power Electronics and Design (ISLPED'10), pages 189–194. Austin, TX, USA, 2010.

[28] DDBJ Sequence Read Archive (DRA). `https://www.ddbj.nig.ac.jp/dra`. [Last visited: October 2018].

[29] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[30] E. Dede, Z. Fadika, M. Govindaraju, and L. Ramakrishnan. Benchmarking MapReduce implementations under different application scenarios. *Future Generation Computer Systems*, 36:389–399, 2014.

[31] S. Desrochers, C. Paradis, and V. M. Weaver. A validation of DRAM RAPL power measurements. In *2nd International Symposium on Memory Systems (MEMSYS'16)*, pages 455–470. Alexandria, VA, USA, 2016.

[32] M. Dimitrov, K. Kumar, P. Lu, V. Viswanathan, and T. Willhalm. Memory system characterization of Big Data workloads. In *2013 IEEE International Conference on Big Data (IEEE BigData 2013)*, pages 15–22. Santa Clara, CA, USA, 2013.

[33] dstat: versatile resource statistics tool. `http://dag.wiee.rs/home-made/dstat/`. [Last visited: October 2018].

[34] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *19th ACM International Symposium on High Performance Distributed Computing (HPDC'2010)*, pages 810–818. Chicago, IL, USA, 2010.

[35] R. R. Expósito, J. Veiga, J. González-Domínguez, and J. Touriño. MarDRe: efficient MapReduce-based removal of duplicate DNA reads in the cloud. *Bioinformatics*, 33(17):2762–2764, 2017.

[36] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. MARIANE: using MapReduce in HPC environments. *Future Generation Computer Systems*, 36:379–388, 2014.

[37] Z. Fadika, M. Govindaraju, R. Canon, and L. Ramakrishnan. Evaluating Hadoop for data-intensive scientific operations. In *5th IEEE International Conference on Cloud Computing (CLOUD'12)*, pages 67–74. Honolulu, HI, USA, 2012.

[38] E. Feller, L. Ramakrishnan, and C. Morin. On the performance and energy efficiency of Hadoop deployment models. In *2013 IEEE International Conference on Big Data (IEEE BigData 2013)*, pages 131–136. Santa Clara, CA, USA, 2013.

[39] E. Feller, L. Ramakrishnan, and C. Morin. Performance and energy efficiency of Big Data applications in cloud environments: a Hadoop case study. *Journal of Parallel and Distributed Computing*, 79:80–89, 2015.

[40] B. Feng, J. Lu, Y. Zhou, and N. Yang. Energy efficiency for MapReduce workloads: an in-depth study. In *23rd Australasian Database Conference (ADC'12)*, pages 61–70. Melbourne, Australia, 2012.

[41] B. Fitzpatrick. Distributed caching with Memcached. *Linux Journal*, 124:72–76, 2004.

[42] Flink Gelly: Apache Flink's graph-processing API and library. `https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/gelly/index.html`. [Last visited: October 2018].

[43] FlinkML: machine learning library for Flink. `https://github.com/apache/flink/tree/master/flink-libraries/flink-ml`. [Last visited: October 2018].

[44] Gartner IT glossary: Big Data. `https://www.gartner.com/it-glossary/big-data/`. [Last visited: October 2018].

[45] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: towards an industry standard benchmark for Big Data analytics. In *2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, pages 1197–1208. New York, NY, USA, 2013.

[46] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *6th Workshop on Programming Languages and Operating Systems (PLOS'11)*, pages 7:1–7:5. Cascais, Portugal, 2011.

[47] S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, J. N. Burton, B. J. Walker, T. Sharpe, G. Hall, T. P. Shea, S. Sykes, A. M. Berlin, D. Aird, M. Costello, R. Daza, L. Williams, R. Nicol, A. Gnirke, C. Nusbaum, E. S. Lander, and D. B. Jaffe. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[48] P. González, X. C. Pardo, D. R. Penas, D. Teijeiro, J. R. Banga, and R. Doallo. Using the cloud for parameter estimation problems: comparing Spark vs MPI with a case-study. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017)*, pages 797–806. Madrid, Spain, 2017.

[49] J. González-Domínguez and B. Schmidt. ParDRe: faster parallel duplicated reads removal tool for sequencing studies. *Bioinformatics*, 32(10):1562–1564, 2016.

[50] Grid'5000: large-scale resource provisioning network. `https://www.grid5000.fr`. [Last visited: October 2018].

[51] GridMix: a benchmark for Hadoop clusters. `https://hadoop.apache.org/docs/stable1/gridmix.html`. [Last visited: October 2018].

[52] J. Grimmer. We are all social scientists now: how Big Data, machine learning, and causal inference work together. *PS: Political Science & Politics*, 48(1):80–83, 2015.

[53] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[54] R. Gu, X. Yang, J. Yan, Y. Sun, B. Wang, C. Yuan, and Y. Huang. SHadoop: improving MapReduce performance by optimizing job execution mechanism in

Hadoop clusters. *Journal of Parallel and Distributed Computing*, 74(3):2166–2179, 2014.

[55] O. Gutsche, M. Cremonesi, P. Elmer, B. Jayatilaka, J. Kowalkowski, J. Pivarski, S. Sehrish, C. M. Surez, A. Svyatkovskiy, and N. Tran. Big Data in HEP: a comprehensive use case study. *Journal of Physics: Conference Series*, 898(7):072012, 2017.

[56] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 313–326. San Diego, CA, USA, 2005.

[57] High-Performance Big Data (HiBD) project. `http://hibd.cse.ohio-state.edu/`. [Last visited: October 2018].

[58] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, pages 295–308. Boston, MA, USA, 2011.

[59] P. Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[60] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In *26th IEEE International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. Long Beach, CA, USA, 2010.

[61] IBTA, InfiniBand Trade Association. `http://www.infinibandta.org`. [Last visited: October 2018].

[62] M. H. Iqbal and T. R. Soomro. Big Data analysis: Apache Storm perspective. *International Journal of Computer Trends and Technology*, 19(1):9–14, 2015.

[63] P. Jakovits and S. N. Srirama. Evaluating MapReduce frameworks for iterative scientific computing applications. In *2014 International Conference on High Performance Computing & Simulation (HPCS'14)*, pages 226–233. Bologna, Italy, 2014.

[64] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo. Characterizing data analysis workloads in data centers. In *2013 IEEE International Symposium on Workload Characterization (IISWC'13)*, pages 66–76. Portland, OR, USA, 2013.

[65] Z. Jia, J. Zhan, L. Wang, R. Han, S. A. McKee, Q. Yang, C. Luo, and J. Li. Characterizing and subsetting Big Data workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC'14)*, pages 191–201. Raleigh, NC, USA, 2014.

[66] J. Jin, J. Gubbi, S. Marusic, and M. Palaniswami. An information framework for creating a smart city through Internet of Things. *IEEE Internet of Things Journal*, 1(2):112–121, 2014.

[67] K. Kambatla, G. Kollias, V. Kumar, and A. Grama. Trends in Big Data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[68] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: a peta-scale graph mining system - implementation and observations. In *9th IEEE International Conference on Data Mining (ICDM'09)*, pages 229–238. Miami, FL, USA, 2009.

[69] K. N. Khan, M. A. Hoque, T. Niemi, Z. Ou, and J. K. Nurminen. Energy efficiency of large scale graph processing platforms. In *2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2016)*, pages 1287–1294. Heidelberg, Germany, 2016.

[70] K. Kim, K. Jeon, H. Han, S.-G. Kim, H. Jung, and H. Y. Yeom. MRBench: a benchmark for MapReduce framework. In *14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*, pages 11–18. Melbourne, Australia, 2008.

[71] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: a modern,

open-source SQL engine for Hadoop. In *7th Biennial Conference on Innovative Data Systems Research (CIDR'15)*, pages 28:1–28:10. Asilomar, CA, USA, 2015.

[72] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *6th International Workshop on Networking Meets Databases (NetDB II)*, pages 1–7. Athens, Greece, 2011.

[73] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: stream processing at scale. In *2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 239–250. Melbourne, Australia, 2015.

[74] B. Lange and T. Nguyen. A Hadoop use case for engineering data. In *12th International Conference on Cooperative Design, Visualization and Engineering (CDVE'15)*, pages 134–141. Mallorca, Spain, 2015.

[75] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. SparkBench: a Spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, 20(3):2575–2589, 2017.

[76] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo. A performance study of Big Data on small nodes. *Proceedings of the VLDB Endowment*, 8(7):762–773, 2015.

[77] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. *Proceedings of the VLDB Endowment*, 9(12):936–947, 2016.

[78] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. DataMPI: extending MPI to Hadoop-like Big Data computing. In *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS'14)*, pages 829–838. Phoenix, AZ, USA, 2014.

[79] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of Apache Spark with RDMA and its benefits on various workloads. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016)*, pages 253–262. Washington, DC, USA, 2016.

[80] C. Luo, J. Zhan, Z. Jia, L. Wang, G. Lu, L. Zhang, C.-Z. Xu, and N. Sun. Cloudrank-D: benchmarking and ranking cloud computing systems for data processing applications. *Frontiers of Computer Science*, 6(4):347–362, 2012.

[81] M. Malik, S. Rafatirah, A. Sasan, and H. Homayoun. System and architecture level characterization of Big Data applications on big and little core server architectures. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015)*, pages 85–94. Santa Clara, CA, USA, 2015.

[82] 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'16)*, pages 433–442. Taipei, Taiwan, 2016.

[83] S. Markidis, I. B. Peng, J. L. Träff, V. B. Antoine Rougier, R. Machado, M. Rahn, A. Hart, D. Holmes, M. Bull, and E. Laure. The EPiGRAM project: preparing parallel programming models for Exascale. In *Workshop on Exascale Multi/Many Core Computing Systems (E-MuCoCoS)*, pages 56–68. Frankfurt, Germany, 2016.

[84] A. J. Martin. Towards an energy complexity of computation. *Information Processing Letters*, 77(2–4):181–187, 2001.

[85] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang, and W. Shi. Energy-aware scheduling of MapReduce jobs for Big Data applications. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2720–2733, 2015.

[86] Mellanox Technologies: Hadoop. `http://www.mellanox.com/page/hadoop`. [Last visited: October 2018].

[87] D. Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 239:76–91, 2014.

[88] Microsoft Azure: cloud computing platform & services. `https://azure.microsoft.com`. [Last visited: October 2018].

[89] C. Mobius, W. Dargie, and A. Schill. Power consumption estimation models for processors, virtual machines, and servers. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1600–1614, 2014.

[90] MongoDB: document-oriented NoSQL database. https://www.mongodb.com/. [Last visited: October 2018].

[91] C. Negru, M. Mocanu, V. Cristea, S. Sotiriadis, and N. Bessis. Analysis of power consumption in heterogeneous virtual machine environments. *Soft Computing*, 21(16):4531–4542, 2017.

[92] K. Nguyen, L. Fang, G. H. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: a high-performance Big-Data-friendly garbage collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 349–365. Savannah, GA, USA, 2016.

[93] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[94] Oprofile: a system profiler for Linux. http://oprofile.sourceforge.net. [Last visited: October 2018].

[95] J. Park, M. Han, and W. Baek. Quantifying the performance impact of large pages on in-memory Big-Data workloads. In *2016 IEEE International Symposium on Workload Characterization (IISWC'16)*, pages 1–10. Providence, RI, USA, 2016.

[96] PigMix: queries for testing Pig performance. https://cwiki.apache.org/confluence/display/PIG/PigMix. [Last visited: October 2018].

[97] Project Tungsten. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. [Last visited: October 2018].

[98] S. Qian, G. Wu, J. Huang, and T. Das. Benchmarking modern distributed streaming platforms. In *2016 IEEE International Conference on Industrial Technology (ICIT 2016)*, pages 592–598. Taipei, Taiwan, 2016.

[99] RAPL read tool. https://github.com/LPD-EPFL/raplread. [Last visited: October 2018].

[100] Redis: open-source in-memory database. `https://redis.io/`. [Last visited: October 2018].

[101] D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: the evolution of data to life-critical. `https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf`. [Last visited: October 2018].

[102] M. Rostanski, K. Grochla, and A. Seman. Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. In *2014 Federated Conference on Computer Science and Information Systems (FedCSIS'14)*, pages 879–884. Warsaw, Poland, 2014.

[103] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: a unifying framework for modeling and building data processing applications. In *2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 1357–1369. Melbourne, Australia, 2015.

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

[105] A. Sangroya, D. Serrano, and S. Bouchenak. MRBS: towards dependability benchmarking for Hadoop MapReduce. In *18th International Euro-Par Conference on Parallel Processing Workshops (Euro-Par'12)*, pages 3–12. Rhodes Island, Greece, 2012.

[106] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: MapReduce vs. Spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[107] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: increased performance for in-memory Hadoop jobs. *Proceedings of the VLDB Endowment*, 5(12):1736–1747, 2012.

[108] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed

File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010)*, pages 1–10. Incline Village, NV, USA, 2010.

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

[110] Spark GraphX: Apache Spark's API for graphs and graph-parallel computation. `http://spark.apache.org/graphx/`. [Last visited: October 2018].

[111] Spark MLlib: Apache Spark's scalable machine learning library. `http://spark.apache.org/mllib/`. [Last visited: October 2018].

[112] Spark Streaming. `https://spark.apache.org/streaming/`. [Last visited: October 2018].

[113] S. Sumimoto, Y. Ajima, K. Saga, T. Nose, N. Shida, and T. Nanri. The design of advanced communication to reduce memory usage for Exascale systems. In *12th International Meeting on High Performance Computing for Computational Science (VECPAR'16)*. Porto, Portugal, 2016.

[114] Sun Microsystems. Memory management in the Java HotSpot™ virtual machine. `http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf`. [Last visited: October 2018].

[115] Y. Tang, H. Guo, T. Yuan, Q. Wu, X. Li, C. Wang, X. Gao, and J. Wu. OEHadoop: accelerate Hadoop applications by co-designing Hadoop with data center network. *IEEE Access*, 6:25849–25860, 2018.

[116] M. Tatineni, X. Lu, D. Choi, A. Majumdar, and D. K. Panda. Experiences and benefits of running RDMA-Hadoop and Spark on SDSC Comet. In *5th Annual Conference on Diversity, Big Data, and Science at Scale (XSEDE'16)*, pages 23:1–23:5. Miami, FL, USA, 2016.

[117] TeraSort for Apache Spark and Flink. `https://github.com/eastcirclek/terasort`. [Last visited: October 2018].

[118] The jstat utility. http://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr017.html. [Last visited: October 2018].

[119] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a Petabyte scale data warehouse using Hadoop. In *26th IEEE International Conference on Data Engineering (ICDE 2010)*, pages 996–1005. Long Beach, CA, USA, 2010.

[120] N. Tiwari, S. Sarkar, U. Bellur, and M. Indrawan. An empirical study of Hadoop's energy efficiency on a HPC cluster. In *International Conference on Computational Science (ICCS'14)*, pages 62–72. Cairns, Australia, 2014.

[121] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th ACM Symposium on Cloud Computing (SoCC'13)*, pages 5:1–5:16. Santa Clara, CA, USA, 2013.

[122] J. Veiga, J. Enes, R. R. Expósito, and J. Touriño. BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks. *Future Generation Computer Systems*, 86:565–581, 2018.

[123] J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Touriño. Performance evaluation of Big Data frameworks for large-scale data analytics. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016)*, pages 424–431. Washington, DC, USA, 2016.

[124] J. Veiga, R. R. Expósito, B. Raffin, and J. Touriño. Optimization of real-world MapReduce applications with Flame-MR: practical use cases. 2018. (Submitted for journal publication).

[125] J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. MREv: an automatic MapReduce Evaluation tool for Big Data workloads. In *International Conference on Computational Science (ICCS'15)*, pages 80–89. Reykjavík, Iceland, 2015.

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

[127] J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Flame-MR: an event-driven architecture for MapReduce applications. *Future Generation Computer Systems*, 65:46–56, 2016.

[128] J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Enhancing in-memory efficiency for MapReduce-based data processing. *Journal of Parallel and Distributed Computing*, 120:323–338, 2018.

[129] J. Veiga, R. R. Expósito, and J. Touriño. Performance evaluation of Big Data analysis. In S. Sakr and A. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–6. Springer International Publishing, Cham, 2018.

[130] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: a Big Data benchmark suite from Internet services. In *20th IEEE International Symposium on High-Performance Computer Architecture (HPCA'14)*, pages 488–499. Orlando, FL, USA, 2014.

[131] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 57:1–57:10. Seattle, WA, USA, 2011.

[132] M. Wasi-Ur-Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance RDMA-based design of Hadoop MapReduce over InfiniBand. In *27th IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW'13)*, pages 1908–1917. Boston, MA, USA, 2013.

[133] V. M. Weaver. Linux perf_event features and overhead. In *2nd International Workshop on Performance Analysis of Workload Optimized Systems (FastPath'13)*, page 80. Austin, TX, USA, 2013.

[134] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring energy and power with PAPI. In *41st International Conference on Parallel Processing Workshops (ICPPW'12)*, pages 262–268. Pittsburgh, PA, USA, 2012.

[135] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable Internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[136] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, pages 13–24. New York, NY, USA, 2013.

[137] W. Xiong, Z. Yu, Z. Bei, J. Zhao, F. Zhang, Y. Zou, X. Bai, Y. Li, and C. Xu. A characterization of Big Data benchmarks. In *2013 IEEE International Conference on Big Data (IEEE BigData 2013)*, pages 118–125. Santa Clara, CA, USA, 2013.

[138] L. D. Xu, E. L. Xu, and L. Li. Industry 4.0: state of the art and future trends. *International Journal of Production Research*, 56(8):2941–2962, 2018.

[139] P. Xuan, W. B. Ligon, P. K. Srimani, R. Ge, and F. Luo. Accelerating Big Data analytics on HPC clusters using two-level storage. *Parallel Computing*, 61:18–34, 2017.

[140] D. Yan, X.-S. Yin, C. Lian, X. Zhong, X. Zhou, and G.-S. Wu. Using memory in the right way to accelerate Big Data processing. *Journal of Computer Science and Technology*, 30(1):30–41, 2015.

[141] D. Yang, X. Zhong, D. Yan, F. Dai, X. Yin, C. Lian, Z. Zhu, W. Jiang, and G. Wu. NativeTask: a Hadoop compatible framework for high performance. In *2013 IEEE International Conference on Big Data (IEEE BigData 2013)*, pages 94–101. Santa Clara, CA, USA, 2013.

[142] T. Yoo, M. Yim, I. Jeong, Y. Lee, and S.-T. Chun. Performance evaluation of in-memory computing on scale-up and scale-out cluster. In *8th International*

Conference on Ubiquitous and Future Networks (ICUFN 2016), pages 456–461. Vienna, Austria, 2016.

[143] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief. LERC: coordinated cache management for data-parallel systems. In *2017 IEEE Global Communications Conference (GLOBECOM 2017)*, pages 1–6. Singapore, 2017.

[144] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: a fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pages 15–28. San Jose, CA, USA, 2012.

[145] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for Big Data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[146] F. Zhang, J. Cao, S. U. Khan, K. Li, and K. Hwang. A task-level adaptive MapReduce framework for real-time streaming data in healthcare applications. *Future Generation Computer Systems*, 43-44:149–160, 2015.

[147] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: a distributed computing framework for iterative computation. *Journal of Grid Computing*, 10(1):47–68, 2012.

[148] Z. Zhang, K. Barbary, F. A. Nothaft, E. R. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter. Kira: processing astronomy imagery using Big Data technology. *IEEE Transactions on Big Data*. (In press). doi: `10.1109/TBDATA.2016.2599926`.

Appendix A

Resumen extendido en castellano

El uso de tecnologías Big Data para el procesamiento de datos a gran escala se está extendiendo a gran velocidad, transformando la forma en la que obtenemos información valiosa de grandes volúmenes de datos. A medida que el tamaño de los datos aumenta, su análisis resulta más exigente para los sistemas de computación actuales. Esta situación pone en el punto de mira el rendimiento y escalabilidad de los entornos de procesamiento de datos masivos, lo que a su vez requiere nuevas maneras de evaluarlos y optimizarlos. La presente Tesis, “Evaluación y Optimización del Procesamiento Big Data en Entornos de Computación de Altas Prestaciones”, aborda estas cuestiones por medio del diseño e implementación de nuevas herramientas con el objetivo de ayudar a desarrolladores y usuarios a identificar y aliviar los posibles problemas de rendimiento.

Introducción

La cantidad de datos que generamos y recopilamos se incrementa día a día de manera exponencial. De hecho, la International Data Corporation (IDC) prevé que generaremos 163 ZB de datos en el año 2025, diez veces los 16.1 ZB generados en 2016 [101]. El término Big Data se refiere al uso de estos grandes volúmenes de datos para obtener información útil, aportando soluciones a problemas analíticos en múltiples campos como ciudades inteligentes [66], ciencias sociales [52], medici-

na [146], industria [138] y muchos más. Los principales retos a los que se enfrentan las tecnologías Big Data son definidos por Gartner, Inc. como volumen, variedad y velocidad [44]. Estos tres retos son cada vez más difíciles de gestionar, ya que ejercen una gran presión en el rendimiento de los sistemas de procesamiento de datos.

La creciente adopción del Big Data se ha fomentado por el cambio en la forma en la que se definen las operaciones de procesamiento de datos. La publicación del modelo de programación MapReduce por parte de Google [29] sentó la base de un nuevo paradigma de computación que se centra en trasladar la computación al lugar donde se almacenan los datos en vez de trasladar los datos al lugar donde se procesan. MapReduce se utiliza popularmente en entornos de procesamiento Big Data, principalmente Apache Hadoop [8]. Otros entornos como Apache Spark [145] o Apache Flink [7] también utilizan parte de su semántica, aunque permiten realizar un conjunto de transformaciones más amplio. Todos estos entornos procesan de forma distribuida grandes volúmenes de datos en sistemas clúster y cloud que pueden tener hasta miles de nodos [67]. A medida que la cantidad de datos a procesar aumenta, la capacidad de computación necesaria para ello alcanza niveles difíciles de gestionar por los sistemas actuales. Esta situación obliga a desarrolladores y usuarios de Big Data a hacer un esfuerzo considerable para identificar y optimizar los problemas de rendimiento que puedan existir en las tecnologías y sistemas actuales, con el fin de mantener los tiempos de ejecución de las aplicaciones dentro de unos límites aceptables, a la vez que se garantiza la calidad de los resultados.

Determinar los factores que limitan el rendimiento de los entornos Big Data no es una tarea trivial, ya que se ven afectados por una gran variedad de aspectos como su diseño e implementación, la planificación de las tareas, la distribución de la carga de trabajo o la arquitectura del sistema. Esto ha supuesto un gran estímulo para la actividad investigadora en torno a la evaluación del rendimiento y el uso de recursos de entornos Big Data. Sin embargo, los resultados de estos estudios son difíciles de extrapolar a casos de uso concretos sin antes obtener información empírica, debido al gran número de factores que influyen en su comportamiento. Por lo tanto, identificar problemas de rendimiento de una carga de trabajo requiere un profundo análisis experimental. Ya que los usuarios de Big Data provienen de campos muy diferentes, en ocasiones sin conocimiento acerca del comportamiento a bajo nivel de los entornos, pueden no contar ni con las habilidades ni con el tiempo

necesario para realizar este tipo de tareas adecuadamente.

Otra problemática a destacar está relacionada con las limitaciones de rendimiento que los entornos ejercen con frecuencia sobre las aplicaciones. Un ejemplo claro de ello se puede encontrar en gran cantidad de aplicaciones Hadoop que se han desarrollado desde su liberación en 2007, alguna de ellas como resultado de meses o años de desarrollo. Aunque Hadoop ha seguido evolucionando y mejorando durante este tiempo, algunas decisiones de diseño iniciales siguen perjudicando su rendimiento general. Para aliviar este problema han surgido entornos más modernos, como Spark o Flink, mejorando el rendimiento de Hadoop y ofreciendo APIs más nuevas y amplias. La mayoría de aplicaciones que se han portado a estas APIs han obtenido grandes mejoras de rendimiento.

Sin embargo, un gran número de aplicaciones aún utiliza Hadoop para el procesamiento de datos, lo que reduce el rendimiento que pueden alcanzar. Adaptarlas para usar Spark o Flink no siempre es asumible debido al excesivo esfuerzo de programación y testeo que se requiere. Así mismo, la mejora de rendimiento que se puede obtener es a priori desconocida, ya que depende de las características de la aplicación. Esto puede provocar que el rendimiento sea igual o incluso peor cuando se ejecuta con Spark o Flink, lo que supondría un gran gasto de tiempo y recursos humanos que podrían haberse empleado mejor en otras tareas. En algunos casos, el código fuente ni siquiera estaría disponible, lo que impediría su modificación.

El propósito de esta Tesis es paliar algunas de las principales dificultades de evaluar y optimizar aplicaciones Big Data, proporcionando nuevas herramientas para ayudar a los usuarios a obtener información acerca del comportamiento de los entornos Big Data y optimizar aplicaciones ya existentes sin comprometer la compatibilidad. Por un lado, la evaluación del rendimiento de los entornos se ha abordado desarrollando una nueva herramienta de evaluación, Big Data Evaluator (BDEv), que permite evaluar de manera automática los entornos de procesamiento Big Data más populares. Por otro lado, se ha propuesto el entorno Flame-MR para permitir la optimización transparente de aplicaciones Hadoop, utilizando computación en memoria para rediseñar la manera en la que se procesan los datos.

Objetivos y Metodología de Trabajo

Los principales objetivos de esta Tesis se listan a continuación.

1. Desarrollo de una herramienta automática de evaluación de entornos de procesamiento Big Data.
    - Configuración y despliegue automático de entornos.
    - Soporte para distintos benchmarks de diferentes tipos.
    - Evaluación con múltiples métricas.
    - Fácil recolección de resultados (p.e. generación automática de gráficas).
2. Diseño e implementación de un nuevo entorno MapReduce para el procesamiento de datos en memoria.
    - Reemplazo transparente de la arquitectura de Hadoop.
    - Aprovechamiento de los recursos del sistema (p.e. CPU, memoria).
    - Solapamiento eficiente del procesamiento.
    - Compatibilidad entre aplicaciones.
3. Evaluación exhaustiva del rendimiento del nuevo entorno.
    - Evaluación en plataformas cloud y sistemas de computación de altas prestaciones.
    - Ejecución de benchmarks estándar y aplicaciones de uso real.
    - Caracterización y análisis en profundidad de las aplicaciones.

Estos objetivos se han llevado a cabo utilizando una metodología de trabajo clásica en investigación e ingeniería: análisis, diseño, implementación y evaluación. Esta metodología se ha aplicado al primer y segundo objetivo, que se corresponden con los ciclos de desarrollo de la Tesis. Para abordar el primer objetivo se ha estudiado el estado del arte actual referente a la evaluación de entornos de procesamiento de datos, identificando los problemas más comunes que se encuentran a la hora de realizar evaluaciones experimentales en sistemas Big Data. Los resultados

de este análisis se han utilizado como base para diseñar la herramienta MapReduce Evaluator (MREv), que más tarde evolucionó a una herramienta más completa, Big Data Evaluator (BDEv). Una vez completada su implementación, se ha testado adecuadamente utilizando múltiples entornos, aplicaciones y sistemas.

Para alcanzar el segundo objetivo se ha desarrollado un nuevo entorno llamado Flame-MR. Utilizando los resultados obtenidos por BDEv en la primera parte de la Tesis, se han identificado diversos puntos débiles de rendimiento para los que Flame-MR ha sido específicamente diseñado. Este diseño se ha realizado de manera modular, implementando y testando los componentes software con una aproximación iterativa. Una vez obtenida la versión inicial del entorno, se han desarrollado diversas técnicas de gestión de memoria para mejorar el rendimiento. La versión final de Flame-MR se ha evaluado en sistemas cloud y clúster de altas prestaciones, ejecutando benchmarks estándar y aplicaciones de uso real, para alcanzar el tercer objetivo de la Tesis.

Conclusiones

La investigación llevada a cabo en esta Tesis ha permitido alcanzar los objetivos de manera satisfactoria. En primer lugar, el desarrollo de la herramienta BDEv para evaluar entornos Big Data ha permitido facilitar la obtención de resultados experimentales con el fin de analizar en profundidad el comportamiento de dichos entornos. BDEv gestiona de manera automática la configuración y despliegue en modo clúster de los entornos, la generación de los conjuntos de datos de entrada, la ejecución de los benchmarks y aplicaciones y la extracción de diversas métricas de rendimiento. Estas métricas no se limitan a tiempos de ejecución, sino que también incluyen escalabilidad, uso de recursos, eficiencia energética y comportamiento a nivel de microarquitectura. El uso de BDEv facilita al usuario la configuración de los entornos de manera homogénea, lo que a su vez permite una comparación más justa entre ellos. Su utilidad se ha demostrado en diversos casos prácticos, ya que la información obtenida ha mostrado ser de gran valor para el análisis de entornos Big Data como Hadoop, Spark y Flink.

Con respecto a la optimización del procesamiento de datos, esta Tesis propone

Flame-MR, un entorno MapReduce que acelera aplicaciones Hadoop ya existentes sin necesidad de modificar el código fuente escrito por el usuario. Flame-MR se basa en una arquitectura dirigida por eventos que paraleliza eficientemente las operaciones de procesamiento y los movimientos de datos, aprovechando los recursos computacionales, como CPU y memoria, de manera más eficiente. El diseño de Flame-MR también reduce el número de copias en memoria que realiza Hadoop, además de utilizar algoritmos eficientes para el ordenamiento y fusión de datos.

Debido a la gran importancia del uso de la memoria en las aplicaciones Big Data, se han implementado diversas técnicas de optimización de la gestión de la memoria con el objetivo de reutilizar los buffers empleados para el procesamiento de los datos. De esta manera se consiguen evitar operaciones innecesarias de reserva de memoria, mejorando el rendimiento de las aplicaciones. También se ha hecho hincapié en reducir el tráfico de disco y red en la medida de lo posible, para lo que se han cacheado en memoria los resultados intermedios. En vez de hacer uso del sistema de ficheros HDFS empleado por Hadoop, los datos de salida se han mantenido en estructuras en memoria para ser procesados por subsiguientes trabajos MapReduce, lo que mejora significativamente el rendimiento de los algoritmos iterativos.

La mejora de rendimiento obtenida por Flame-MR se ha evaluado en diferentes escenarios y sistemas. El uso de benchmarks estándar ha facilitado la comparación entre los resultados de rendimiento de distintos entornos. Utilizando estos benchmarks, Flame-MR reduce el tiempo de ejecución de Hadoop en un 48 % de media, además de obtener resultados muy competitivos con respecto a Spark. Aunque Spark consigue mejor rendimiento en benchmarks iterativos, es superado por Flame-MR al ejecutar benchmarks intensivos en entrada/salida como Sort y PageRank, con el beneficio adicional de no requerir la reescritura del código fuente de las aplicaciones Hadoop, como sí ocurre con Spark.

Flame-MR también se ha evaluado utilizando varias aplicaciones extraídas de casos de uso reales, incluyendo consultas analíticas a bases de datos (VELaSSCo), corrección de errores en conjuntos de datos genómicos (CloudRS) y eliminación de secuencias de ADN duplicadas en conjuntos de datos genómicos (MarDRe). Estas aplicaciones, escritas originalmente para Hadoop, presentan una aceleración de entre un 40 % y un 90 % cuando se ejecutan con Flame-MR, lo que supone que su tiempo de ejecución se reduce hasta un orden de magnitud en algunos casos. Estos resultados,

junto con los obtenidos usando benchmarks estándar, demuestran claramente los beneficios de Flame-MR sobre Hadoop en términos de rendimiento.

Trabajo Futuro

La propuesta de nuevas herramientas de evaluación y optimización de entornos y aplicaciones Big Data sienta la base para líneas de trabajo futuras que se beneficien del trabajo llevado a cabo durante esta Tesis. A medida que los ecosistemas Big Data se hacen más heterogéneos, tanto en términos de hardware como de software, es necesario aumentar el esfuerzo para encontrar nuevas técnicas de evaluación que permitan comparar de manera precisa distintos sistemas y paradigmas.

Aunque el tiempo de ejecución se utiliza comúnmente como principal métrica de rendimiento para trabajos batch, no se puede aplicar en todos los escenarios (p.e. procesamiento streaming), lo que dificulta la comparación de rendimiento entre diferentes paradigmas. Por lo tanto, es necesario proponer métricas estándar más actualizadas, análogas a las utilizadas en paradigmas paralelos más tradicionales (p.e. operaciones en punto flotante por segundo, FLOPS). Estas nuevas métricas deben facilitar la extrapolación de resultados cuando se varían parámetros experimentales como el tamaño de los datos de entrada o características del sistema como el número de nodos del clúster. Aunque la cantidad de datos procesados por unidad de tiempo (MB/s) puede ser una aproximación inicial para tal métrica, no siempre es adecuada ya que el tamaño de los datos de entrada puede variar entre escenarios debido a las especificaciones del formato. Por otro lado, las aplicaciones iterativas no emplean todo el tiempo de ejecución procesando los datos de entrada. Estas cuestiones deben tenerse en cuenta cuidadosamente para elaborar una propuesta sólida de métricas estándar de rendimiento.

A medida que los entornos evolucionan con el tiempo, se está haciendo más compleja su utilización y configuración. Esta situación requiere nuevas herramientas y métodos de evaluación que cooperen con los entornos de manera activa. Además de extraer métricas de evaluación como uso de recursos o eficiencia energética, deben ser capaces de utilizar esa información para mejorar el rendimiento. Analizando los datos recogidos, el proceso de evaluación debe retroalimentarse para ajustar la

configuración de los entornos o adaptar los recursos del sistema a las necesidades de las aplicaciones en tiempo real.

En la actualidad, la optimización del procesamiento de datos está orientada a mejorar el rendimiento de un determinado entorno, aliviando problemas existentes en su diseño. Sin embargo, la aplicabilidad de este enfoque se ve claramente limitada al uso de ese entorno en concreto. Las nuevas optimizaciones deben aspirar a buscar soluciones a problemas de rendimiento comunes que se puedan encontrar en múltiples entornos. Ejemplos de ello son ineficiencias en la serialización de datos o en la planificación de las tareas. Encontrar soluciones de diseño eficientes para estas cuestiones puede aportar a los desarrolladores guías de actuación de gran utilidad a la hora de diseñar nuevos entornos de procesamiento de datos u optimizar los ya existentes.

Las APIs de programación que ofrecen los entornos de procesamiento Big Data muestran muchas similitudes entre sí. Por ejemplo, Spark y Flink tienen en común muchas operaciones de transformación de los datos, como *map()* o *filter()*. Por lo tanto, establecer una API estándar de procesamiento de datos sería de gran ayuda para homogeneizar la manera en la que se definen las operaciones en esos entornos. De manera similar a la que el estándar MPI unificó la programación paralela en el ámbito de la computación de altas prestaciones, un modelo de programación estándar para Big Data permitiría múltiples implementaciones de las mismas operaciones, centrándose en conseguir su implementación de manera eficiente mientras se garantiza la compatibilidad entre diferentes entornos de procesamiento.

Principales Contribuciones

Las principales contribuciones originales derivadas de la Tesis son las siguientes:

- Desarrollo de la herramienta BDEv para caracterizar el rendimiento y automatizar la evaluación de entornos Big Data [122, 125].
- Análisis experimental del comportamiento de entornos de procesamiento Big Data [123, 126, 129].

- Desarrollo de Flame-MR para optimizar aplicaciones Hadoop de manera transparente, aprovechando los recursos computacionales de sistemas clúster y cloud de altas prestaciones [127, 128].

- Evaluación exhaustiva de Flame-MR utilizando benchmarks estándar y aplicaciones extraídas de casos de uso reales [35, 124].

Software desarrollado

Las herramientas software desarrolladas en esta Tesis están disponibles públicamente:

- BDEv: herramienta de evaluación automática para entornos de procesamiento Big Data. Disponible en `http://bdev.des.udc.es/`.

- Flame-MR: entorno MapReduce de computación en memoria para optimizar aplicaciones Hadoop de manera transparente. Disponible en `http://flamemr.des.udc.es/`.

- MarDRe: herramienta MapReduce para eliminar secuencias de ADN duplicadas en grandes conjuntos de datos genómicos. Disponible en `http://mardre.des.udc.es/`.

Software registrado

Tres productos software resultado de esta Tesis se han registrado en el Registro de la Propiedad Intelectual:

- J. Veiga, R. R. Expósito, G. L. Taboada, y J. Touriño. Flame-MR: framework MapReduce para computación en memoria, 2018. Número de asiento registral: pendiente. Entidad titular: Universidade da Coruña. País de prioridad: España.

- R. R. Expósito, J. Veiga, J. González-Domínguez, y J. Touriño. MapReduce Duplicate Removal tool: MarDRe, Noviembre 2017. Número de asiento registral: 03/2018/174. Entidad titular: Universidade da Coruña. País de prioridad: España.

- J. Veiga, R. R. Expósito, G. L. Taboada, y J. Touriño. MapReduce Evaluator: MREv, Junio 2016. Número de asiento registral: 03/2016/1054. Entidad titular: Universidade da Coruña. País de prioridad: España. En explotación por Torus Software Solutions S.L. a través del contrato INV13317 desde el 18/12/2017.

Publicaciones de la Tesis

Artículos en revistas

- J. Veiga, R. R. Expósito, B. Raffin, and J. Touriño. Optimization of real-world MapReduce applications with Flame-MR: practical use cases. 2018. (Enviado para publicación en revista).

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Enhancing in-memory efficiency for MapReduce-based data processing. *Journal of Parallel and Distributed Computing*, 120:323–338, 2018. JCR Q2.

- J. Veiga, J. Enes, R. R. Expósito, and J. Touriño. BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks. *Future Generation Computer Systems*, 86:565–581, 2018. JCR Q1 (primer decil).

- R. R. Expósito, J. Veiga, J. González-Domínguez, and J. Touriño. MarDRe: efficient MapReduce-based removal of duplicate DNA reads in the cloud. *Bioinformatics*, 33(17):2762–2764, 2017. JCR Q1 (primer decil).

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Flame-MR: an event-driven architecture for MapReduce applications. *Future Generation Computer Systems*, 65:46–56, 2016. JCR Q1 (primer decil).

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Analysis and evaluation of MapReduce solutions on an HPC cluster. *Computers & Electrical Engineering*, 50:200–216, 2016. JCR Q3.

Congresos internacionales

- J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Touriño. Performance evaluation of Big Data frameworks for large-scale data analytics. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016)*, pages 424–431. Washington, DC, USA, 2016.

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. MREv: an automatic MapReduce Evaluation tool for Big Data workloads. In *International Conference on Computational Science (ICCS'15)*, pages 80–89. Reykjavík, Iceland, 2015.

- J. Veiga, G. L. Taboada, X. C. Pardo, and J. Touriño. The HPS3 service: reduction of cost and transfer time for storing data on clouds. In *16th IEEE International Conference on High Performance Computing and Communications (HPCC'14)*, pages 213–220. Paris, France, 2014.

Capítulos de libro

- J. Veiga, R. R. Expósito, and J. Touriño. Performance evaluation of Big Data analysis. In S. Sakr and A. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–6. Springer International Publishing, Cham, 2018.

Otras publicaciones menores

- J. Veiga, R. R. Expósito, and J. Touriño. Flame-MR: transparent performance improvement of Big Data applications. In *Journée des doctorants de l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII)*. Grenoble, France, 2017.

- J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño. Performance improvement of MapReduce applications using Flame-MR. In *2nd NESUS Winter School & PhD Symposium 2017*. Vibo Valentia, Italy, 2017.