Optimization of real-world MapReduce applications with Flame-MR: practical use cases

Jorge Veiga, Roberto R. Expósito, Bruno Raffin, and Juan Touriño, *Senior Member, IEEE*

***Abstract*—Apache Hadoop is a widely used MapReduce framework for storing and processing large amounts of data. However, it presents some performance issues that hinder its utilization in many practical use cases. Although existing alternatives like Spark or Hama can outperform Hadoop, they require to rewrite the source code of the applications due to API incompatibilities. This paper studies the use of Flame-MR, an in-memory processing architecture for MapReduce applications, to improve the performance of real-world use cases in a transparent way while keeping application compatibility. Flame-MR adapts to the characteristics of the workloads, managing efficiently the use of custom data formats and iterative computations, while also reducing workload imbalance. The experimental evaluation, conducted in high performance clusters and the Microsoft Azure cloud, shows a clear outperformance of Flame-MR over Hadoop. In most cases, Flame-MR reduces the execution times by more than a half.**

***Index Terms*—Big Data, MapReduce, performance optimization, bioinformatics, visualization**

I. Introduction

Nowadays, Big Data applications are employed in a wide range of industrial and research fields to extract meaningful information from large datasets. The adoption of user-friendly technologies like the MapReduce programming model [1] has allowed non-expert programmers to develop large-scale distributed applications without needing to implement low-level functionalities such as data movement and parallelism. This enables them to focus on the actual data processing needed to calculate the desired result.

However, MapReduce applications do not always leverage the computational capabilities of the underlying system. This is often caused by performance drawbacks in the design of popular MapReduce frameworks like Hadoop [2], which incurs limited efficiency on resource usage and data pipelining. Moreover, non-expert programmers can introduce inefficiencies in their applications due to the unawareness of certain framework functionalities (e.g. custom data formats). Although more advanced alternatives like Spark [3] and Hama [4] would allow improving the performance of existing Hadoop workloads, they require to rewrite the source code completely, and so are not always a feasible option. To solve this problem, new in-memory frameworks have been developed to transparently improve the performance of existing Hadoop applications, such as Flame-MR [5], [6]. This framework allows accelerating such applications without changing their source code.

In our previous work [6], the acceleration of Flame-MR was experimentally demonstrated by conducting performance evaluations with synthetic benchmarks. However, the performance of these benchmarks may not always correspond with the one that can be obtained in practical scenarios. This paper aims to overcome this limitation by presenting an in-depth analysis of the performance benefits provided by Flame-MR using three real-world Big Data applications. The main contributions are:

- The identification of significant differences between real-world applications and standard benchmarks, which underscores the importance of using real cases when evaluating Big Data frameworks.
- A detailed description of the techniques used by Flame-MR to adapt to the characteristics of real-world applications, like custom input and output formats and data objects. These techniques provide portability while maintaining performance optimizations.
- An improved version of Flame-MR that includes a new load balancing mode to speed up the processing of skewed datasets.
- The optimization of three real Hadoop applications with Flame-MR to justify its efficiency by alleviating code inefficiencies and load balancing problems. Performance improvements higher than 40% are obtained in all cases.

The rest of the paper is organized as follows: Sections II and III introduce the background and related work, respectively. Sections IV, V and VI analyze the optimization of three use cases based on the MapReduce model. First, Section IV describes the optimization of VELaSSCo, a visualization architecture for simulation data. Second, Section V analyzes CloudRS, a bioinformatics application for error removal in genomic datasets. Third, Section VI presents MarDRe, another genomic application that removes duplicate and near-duplicate reads. Finally, Section VII provides some general conclusions about the results gathered in the previous use cases.

II. Background

The MapReduce programming model was originally proposed by Google in [1]. This model allows developing large-scale Big Data workloads by keeping some implementation details such as 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

Jorge Veiga, Roberto R. Expósito, and Juan Touriño are with the Computer Architecture Group, Universidade da Coruña, Campus de A Coruña, 15071 A Coruña, Spain

Bruno Raffin is with the University Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France

Corresponding author: Jorge Veiga (jorge.veiga@udc.es)

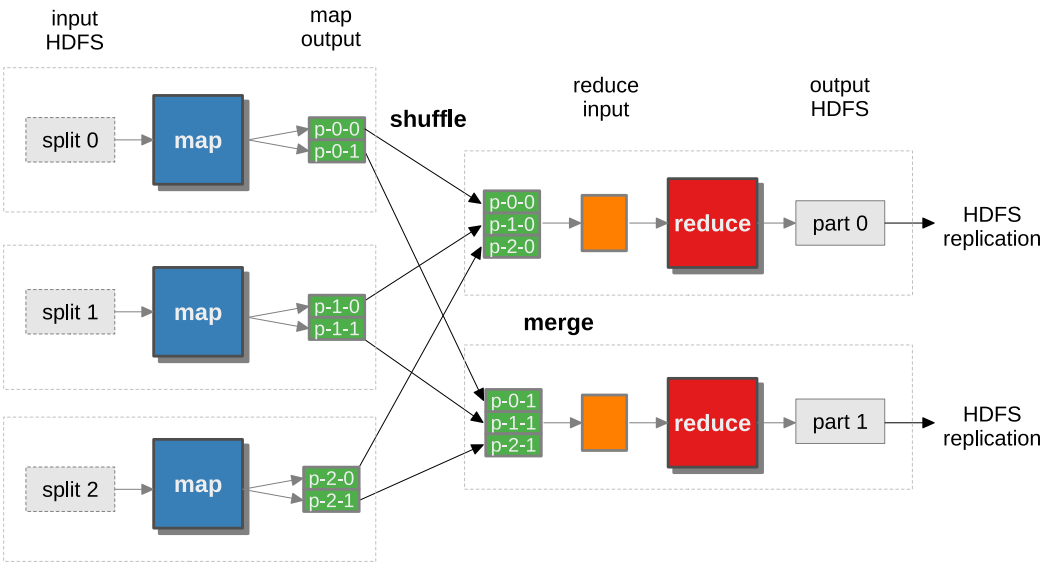


Figure 1: Hadoop data flow with multiple map and reduce tasks

relevant attributes and the reduce function operates them to get a final result.

Nowadays, the de-facto standard implementation of MapReduce is Hadoop [2], an open-source Java-based framework. It mainly consists of two parts, the MapReduce data engine and the Hadoop Distributed File System (HDFS) [7], which distributes the storage of large datasets 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 1. As can be seen, the input dataset stored in HDFS is divided into many splits that are read by map operations to extract the relevant key-value pairs. These key-value pairs are partitioned, sorted by key and sent 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, providing 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 specific 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.

Many applications 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 [3] and Hama [4], which can be used to execute Big Data workloads with a more flexible API and increased performance. However, rewriting existing Hadoop 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.

Our previous work focused on the development of Flame-MR [5], an easy-to-use MapReduce framework that gives solution to this problem by accelerating Hadoop applications without changing the source code defined by the user. Flame-MR replaces transparently the underlying implementation of the Hadoop MapReduce data engine by an in-memory architecture that leverages system resources efficiently.

The operation of Flame-MR is based on the deployment of several Worker processes over the nodes of a cluster. Each Worker is in charge of executing multiple map and reduce operations by using an event-driven architecture shown in Figure 2. The thread pool executes the operations concurrently, scheduling them to pipeline data processing and data movement steps. The data pool allocates memory buffers in an efficient way, reducing the amount of buffer creations [6]. Once the buffers are filled with data, they are stored into in-memory data structures to be processed by subsequent operations.

Performance is further improved by minimizing the connections needed to read and write textual data to HDFS, working with full input splits in memory. Moreover, primitive data types of Hadoop are modified to avoid the use of redundant memory copies, using instead references to the serialized data to optimize pair copies and comparisons. The Hadoop data engine is also accelerated by using efficient sort and merge algorithms. Flame-MR has been assessed by means of synthetic benchmarks, showing significant performance improvement over Hadoop and other Hadoop-based alternatives [5] and also providing competitive results compared to Spark [6].

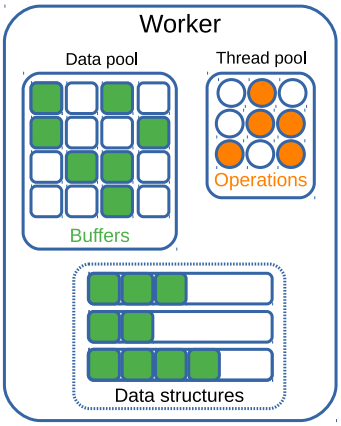


Figure 2: Flame-MR Worker architecture

III. Related work

Many papers in the literature have compared the performance of the MapReduce model when using different data processing engines (e.g., NativeTask [8]), file systems (e.g. MARIANE [9]), and network interconnects (e.g. RDMA-Hadoop [10]). These works generally evaluate their proposals by executing popular Big Data benchmarks like TeraSort or K-Means. However, the lack of performance results with real-world applications 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 [11] is a distributed astronomy image processing toolkit 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 [12] to adapt high energy physics workflows to Spark, obtaining improved usability and performance when compared to other existing sequential implementations like ROOT [13]. 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 [14] to evaluate RDMA-Spark [15], which improves the results of standard Spark with a 1.21× speedup. In the case of Hadoop, the authors of OEHadoop [16] evaluate their proposal by simulating a Facebook job trace extracted from the SWIM project [17]. 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 portable way. Other frameworks that employ in-memory optimizations are NativeTask [8] and M3R [18]. On the one hand, NativeTask is based on a native C++ implementation that replaces the task management of map and reduce functions, while also optimizing the cache awareness of the merge-sort mechanism [19]. As these optimizations are highly dependent on the underlying system, they do not keep portability. On the other hand, M3R is an in-memory framework based on the X10 programming language [20]. Although it accelerates Hadoop workloads by reducing the shuffling overhead and caching intermediate data, it is restricted to Hadoop jobs that can fit in memory. These characteristics prevent the utilization of M3R for real-world Big Data use cases.

As commented in the previous section, the performance benefits of Flame-MR when executing synthetic benchmarks have already been assessed. However, the artificial nature of this kind of workloads makes it difficult to extrapolate these results to practical use cases. This paper provides an in-depth performance analysis of three real-world Hadoop applications, describing the characteristics and challenges of each workload. The main objective is to determine the performance benefits that Flame-MR is able to provide in practice without needing to change the underlying computing paradigm.

IV. VELaSSCo: data visualization queries

This section addresses the optimization of VELaSSCo [21], a Big Data visualization architecture that relies on the MapReduce model to extract information from simulation datasets. More details of this project are provided in Section IV-A, while the main characteristics of the MapReduce workloads are described in Section IV-B. Section IV-C explains the main challenges of running these workloads with Flame-MR to improve their performance. The experimental configuration and performance results are then presented in Sections IV-D and IV-E, respectively. Finally, some concluding remarks are provided in Section IV-F.

A. 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 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 [22], 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. However, 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.

B. 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 mentioned in the previous section, 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 subsystem.

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.

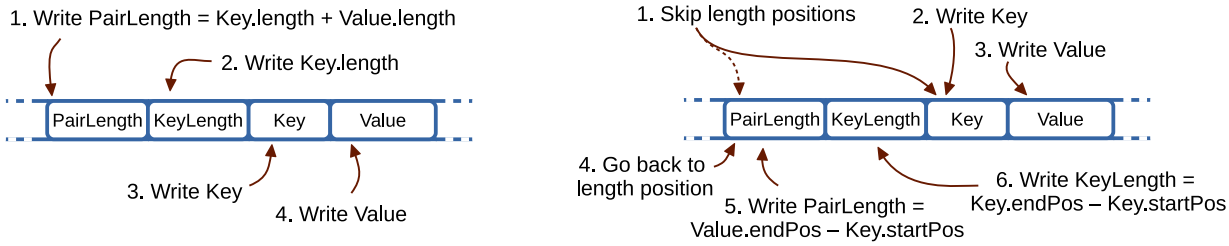
C. Challenges

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 buffers (e.g. 1 MB) allocated in the data pool (see Figure 2). Once the input split is read, the connection to HDFS is closed and the data buffers 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. 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 buffer, a header is added to indicate the pair and key lengths, which will be used to read data without creating 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 defined by Hadoop, which does not provide any information about the length of the objects.



(a) Primitive types (b) Writable objects

Figure 3: Data object serialization in Flame-MR

Table I: Node characteristics of Grid'5000

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

The serialization mechanism of Hadoop primitive data types in Flame-MR is shown in Figure 3a. 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 3b. 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 data buffer to the original position. This mechanism ensures compatibility with all types of Writable objects, while maintaining the in-memory optimizations of Flame-MR.

D. Experimental configuration

This section describes 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 [23]. 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 I for more details).

The experiments have used HBase 1.2.4, Hadoop 2.7.3 and Flame-MR 1.1 (available at http://flamemr.des.udc.es). 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 II, including a brief explanation of each one.

The VELaSSCo queries used in the evaluation are the ones described in Section IV-B. 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 graphs show the median elapsed time of 10 executions, although the standard deviations observed were not significant.

E. Performance results

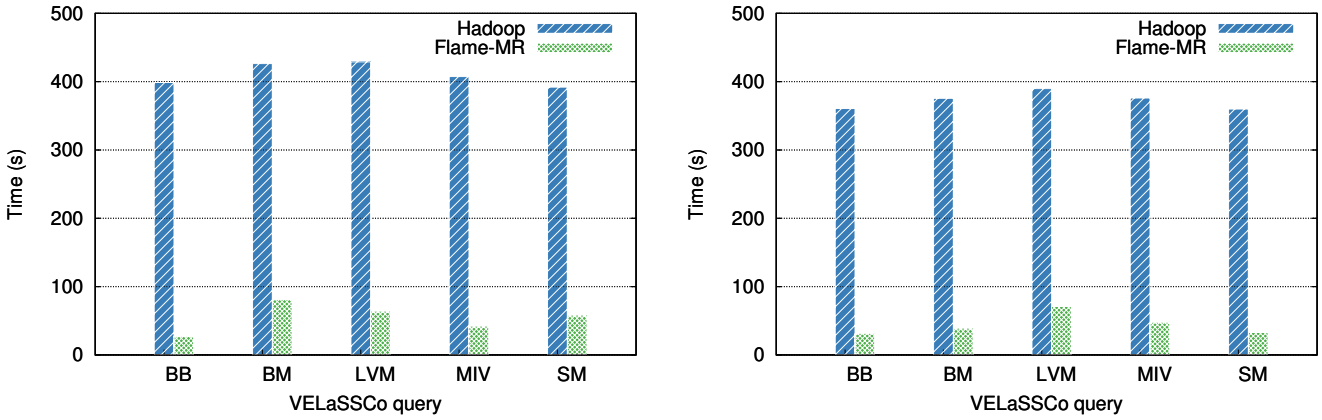
Figures 4a and 4b show the execution times of the VELaSSCo queries using 17 and 25 nodes, respectively. Flame-MR widely outperforms Hadoop with both cluster sizes, showing an average reduction in execution time of 87% with 17 nodes and 88% with 25 nodes. This reduction is due to the more efficient architecture 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 unit time compared to Hadoop, which is reflected in the information counters provided by HBase.

F. Remarks

This section addressed the optimization of analytical queries that 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

Table II: Configuration of the frameworks in Grid'5000

Hadoop MapReduce		
Mapper/Reducer heap size	7.25 GB	Maximum heap size for Mapper/Reducer processes
Mappers per node	8	Maximum number of map tasks per node
Reducers per node	8	Maximum number of reduce tasks per node
Shuffle parallel copies	20	Parallel transfers per Reducer during shuffle phase
IO sort MB	1600 MB	Total amount of memory to use while sorting files
IO sort spill percent	80%	Memory threshold to spill memory data to disk
Flame-MR		
Worker heap size	44 GB	Maximum heap size for Worker processes
Workers per node	2	Number of Worker processes per node
Cores per Worker	8	Number of CPU cores per Worker
Data pool size	30.8 GB	Total amount of memory to store data buffers
Data buffer size	1 MB	Amount of memory allocated to each data buffer
HDFS		
HDFS block size	256 MB	Logical block size for distributed files
Replication factor	3	Number of distributed copies per block



(a) 17 nodes

(b) 25 nodes

Figure 4: Execution times of VELaSSCo queries with Hadoop and Flame-MR

almost one order of magnitude, enhancing the user experience of VELaSSCo.

V. CloudRS: error removal in genomic data

This section addresses the optimization of CloudRS [24], a bioinformatics tool that detects and corrects errors in large genomic datasets, following the same structure as Section IV.

A. 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 [25]. 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.

B. 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 during 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 takes advantage 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 formats 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.

C. Challenges

As explained in the previous section, 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 thread pool to execute the map and reduce functions (see Figure 2). 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 mentioned in Section IV-C. 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 in the previous section 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 implementation 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.

D. Experimental configuration

This section describes the experimental configuration used to evaluate CloudRS with Hadoop and Flame-MR. As genomic applications are executed in many kinds of systems, the evaluation has considered two different scenarios: a private cluster with 9 nodes, Pluton, and a public cloud platform, Microsoft Azure [26], using 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 III and IV, 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 (GbE). The experiments in Azure

Table III: Node characteristics of Pluton

Hardware configuration		Software configuration	
CPU model	2 × Intel Xeon E5-2660 (Sandy Bridge)	OS version	CentOS release 6.8
CPU speed (Turbo)	2.20 GHz (3 GHz)	Kernel	2.6.32-642
Cores	16	Java	Oracle JDK 1.8.0_45
Memory	64 GB DDR3 1600 MHz		
Disk	1 TB HDD		
Network	InfiniBand FDR & GbE		

Table IV: Node characteristics of L16S instances in Azure

Hardware configuration		Software configuration	
CPU model	Intel Xeon E5-2698B v3 (Haswell)	OS version	CentOS release 7.4
CPU speed	2 GHz	Kernel	3.10.0-514
Virtual cores	16	Java	OpenJDK 1.8.0_151
Memory	128 GB		
Disk	2.7 TB SSD		
Network	4 × 10 Gbps Ethernet		

Table V: Configuration of the frameworks in Pluton

Hadoop MapReduce		Flame-MR		HDFS	
Mapper/Reducer heap size	3.4 GB	Worker heap size	11.8 GB	HDFS block size	256 MB
Mappers per node	8	Workers per node	4	Replication factor	3
Reducers per node	8	Cores per Worker	4		
Shuffle parallel copies	20	Data pool size	8.3 GB		
IO sort MB	841 MB	Data buffer size	512 KB		
IO sort spill percent	80%				

Table VI: Configuration of the frameworks in Azure

Hadoop MapReduce		Flame-MR		HDFS	
Mapper/Reducer heap size	6.7 GB	Worker heap size	24 GB	HDFS block size	128 MB
Mappers per node	8	Workers per node	4	Replication factor	3
Reducers per node	8	Cores per Worker	4		
Shuffle parallel copies	20	Data pool size	16.8 GB		
IO sort MB	1704 MB	Data buffer size	512 KB		
IO sort spill percent	80%				

have been carried out using L16S virtual instances located in the West Europe region. These instances have 16 virtual cores per node, 128 GB of memory and a virtual 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 V and VI for Pluton and Azure, respectively. These parameters have been described in Table II. Furthermore, the frameworks have used the IP over InfiniBand (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) [27]. It is composed of 16 million sequences of 100 bases each (5.2 GB in total). The results shown in the following section 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 the Big Data Evaluator (BDEv) tool [28] (version 3.1, available at http://bdev.des.udc.es). This tool allows automating the configuration and deployment of the frameworks and the execution of the workloads.

E. Performance results

Figure 5 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

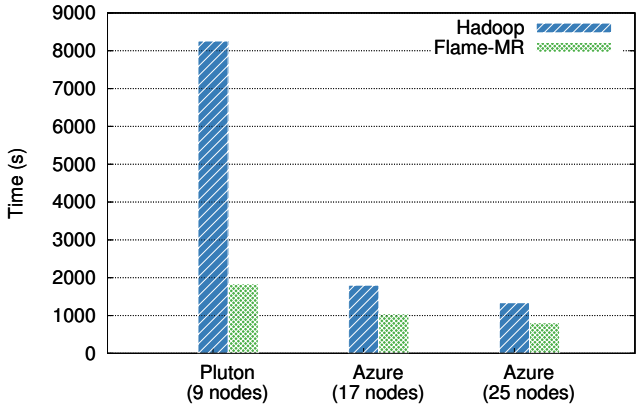


Figure 5: Execution times of CloudRS with Hadoop and Flame-MR

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.

F. 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 CloudRS 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.

VI. MarDRe: duplicate read removal in genome sequencing data

This section addresses the optimization of MarDRe [29], a bioinformatics application that removes duplicate reads in large genomic datasets, following the same structure as Sections IV and V.

A. Overview

As explained in Section V-A, 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 duplicate sequences in genomic datasets stored in HDFS. It is based on a prefix-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 in the next section.

B. 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 [30]. 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. This causes some reducers to have excessive execution times. As a MapReduce job has to wait for all reducers to finish, the load balancing problem in the reduce phase affects the

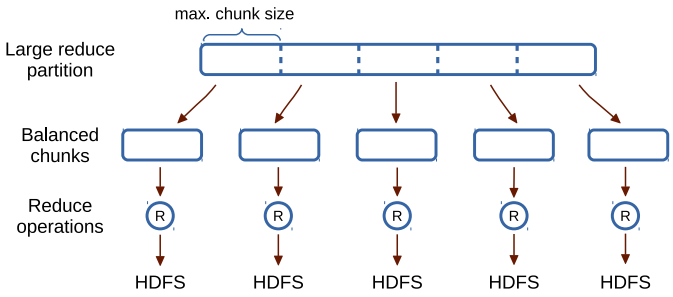


Figure 6: Load balancing mode in Flame-MR

overall performance. The next section discusses how Flame-MR solves this problem in a transparent way.

C. 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, as explained in Section IV-C.

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 time of heavy-loaded reducers is decreased.

Figure 6 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 shown in Section VI-E, the amount of duplicate reads filtered did not vary more than 0.02% when using the load balancing mode.

This mode can be activated by the user via configuration, using a single parameter to indicate the number of chunks in which partitions should be split. By default, this value is set to the number of cores per Worker.

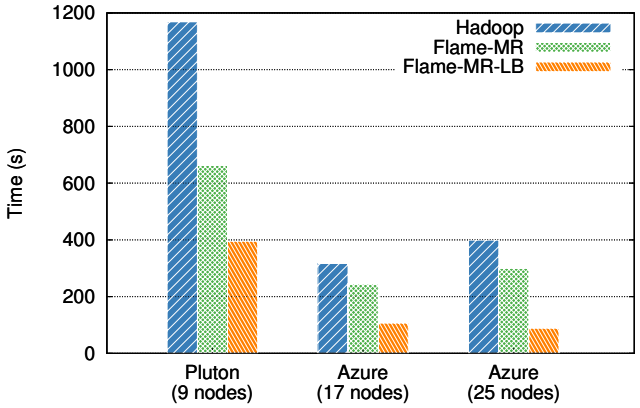


Figure 7: Execution times of MarDRe with Hadoop, Flame-MR and Flame-MR-LB

D. 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 III-VI in Section V-D), and the BDEv tool to conduct the evaluation. However, the computational requirements of CloudRS are significantly higher than those of MarDRe, 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). In the experiments, the number of chunks of the load balancing mode has been tuned for improved performance, splitting each partition in 13 and 9 chunks for Pluton and Azure, respectively.

E. Performance results

Figure 7 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 in Azure. This huge improvement demonstrates the effectiveness of the load balancing mode explained in Section VI-C, together with the efficient in-memory architecture of Flame-MR.

Note that the execution times of Hadoop and Flame-MR in Azure using 25 nodes are higher than with 17 nodes, which is due to the load balancing 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.

Table VII: Load balancing in MarDRe

(a) Pluton (9 nodes)

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

(b) Azure (17 nodes)

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

(c) Azure (25 nodes)

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

In order to provide more information about the load balancing problem of MarDRe, Table VII 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 more nodes in Azure 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 causes the overall execution time to be higher. Flame-MR-LB shows a different behavior. When using 25 nodes, 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.

F. Remarks

This section has shown the benefits of optimizing MarDRe with Flame-MR. Without modifying its source code, it obtains 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.

VII. Conclusion

The MapReduce computing model and Hadoop are commonly used by many applications to extract valuable information from datasets stored in HDFS. Although other alternatives such as Spark can provide improved performance over Hadoop, the effort of adapting existing MapReduce applications to new APIs can be significant (provided that the source code is available). Flame-MR solves this problem by providing huge performance improvements in a transparent way without needing to change the applications.

This paper has shown three different real-world use cases 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 significantly 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 use cases with distinct characteristics, together with the assessment of real-world datasets on different systems, have proved the significant performance benefits provided by Flame-MR over Hadoop.

Acknowledgment

This work was supported by the Ministry of Economy, Industry and Competitiveness of Spain (Project TIN2016-75845-P, AEI/FEDER/EU) and by the FPU Program of the Ministry of Education (grant FPU14/02805). The Grid'5000 testbed used in this paper is supported by Inria, CNRS, RENATER and several French Universities. The authors would like to thank Iván Cores for his contribution to the deployment of VELaSSCo, and also Pierre Neyron and Michael Mercier for their help in the use of the Grid'5000 platform. Also thanks to Microsoft Research for awarding this work with a sponsored Azure account (ref. MS-AZR-0036P).

References

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

[2] Apache Hadoop, http://hadoop.apache.org/, [Last visited: November 2018].

[3] 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*, vol. 59, no. 11, pp. 56–65, 2016.

[4] K. Siddique, Z. Akhtar, E. J. Yoon, Y.-S. Jeong, D. Dasgupta, and Y. Kim, "Apache Hama: an emerging bulk synchronous parallel computing framework for Big Data applications," *IEEE Access*, vol. 4, pp. 8879–8887, 2016.

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

[6] ——, "Enhancing in-memory efficiency for MapReduce-based data processing," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 323–338, 2018.

[7] 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)*, Incline Village, NV, USA, 2010, pp. 1–10.

[8] 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)*, Santa Clara, CA, USA, 2013, pp. 94–101.

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

[10] 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)*, Boston, MA, USA, 2013, pp. 1908–1917.

[11] 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*, 2016, (In press). [Online]. Available: https://doi.org/10.1109/TBDATA.2016.2599926

[12] 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*, vol. 898, no. 7, p. 072012, 2017.

[13] 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*, vol. 389, no. 1-2, pp. 81–86, 1997.

[14] 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)*, Miami, FL, USA, 2016, pp. 23:1–23:5.

[15] 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)*, Washington, DC, USA, 2016, pp. 253–262.

[16] 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*, vol. 6, pp. 25 849–25 860, 2018.

[17] 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*, vol. 5, no. 12, pp. 1802–1813, 2012.

[18] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3R: increased performance for in-memory Hadoop jobs," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.

[19] 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*, vol. 30, no. 1, pp. 30–41, 2015.

[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)*, San Diego, CA, USA, 2005, pp. 519–538.

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

[22] Apache HBase: Hadoop distributed Big Data store, https://hbase.apache.org/, [Last visited: November 2018].

[23] Grid'5000: large-scale resource provisioning network, https://www.grid5000.fr, [Last visited: November 2018].

[24] 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)*, Santa Clara, CA, USA, 2013, pp. 717–722.

[25] 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*, vol. 108, no. 4, pp. 1513–1518, 2011.

[26] Microsoft Azure: cloud computing platform & services, https://azure.microsoft.com, [Last visited: November 2018].

[27] DDBJ Sequence Read Archive (DRA), https://www.ddbj.nig.ac.jp/dra, [Last visited: November 2018].

[28] 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*, vol. 86, pp. 565–581, 2018.

[29] R. R. Expósito, J. Veiga, J. González-Domínguez, and J. Touriño, "MarDRe: efficient MapReduce-based removal of duplicate DNA reads in the cloud," *Bioinformatics*, vol. 33, no. 17, pp. 2762–2764, 2017.

[30] J. González-Domínguez and B. Schmidt, "ParDRe: faster parallel duplicated reads removal tool for sequencing studies," *Bioinformatics*, vol. 32, no. 10, pp. 1562–1564, 2016.