Future Generation Computer Systems 65 (2016) 46–56

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Flame-MR: An event-driven architecture for MapReduce applications

Jorge Veiga *, Roberto R. Expósito, Guillermo L. Taboada, Juan Touriño

Grupo de Arquitectura de Computadores (GAC), Departamento de Electrónica e Sistemas, Facultade de Informática, Universidade da Coruña, Campus de A Coruña, 15071 A Coruña, Spain

HIGHLIGHTS

- Description of Flame-MR, a new MapReduce framework that improves the performance and resource efficiency of Hadoop.
- Flame-MR keeps Hadoop API compatibility in order to avoid source code modifications.
- Performance comparison with Hadoop-based frameworks using representative workloads on an HPC cluster and a cloud platform.
- Flame-MR reduces Hadoop execution times by up to 34% for the selected micro-benchmarks and 54% for the application benchmarks.

ARTICLE INFO

Article history:
Received 4 February 2016
Received in revised form
27 April 2016
Accepted 9 June 2016
Available online 17 June 2016

Keywords:
Big Data
MapReduce
Hadoop
Event-driven architecture
Cloud computing

ABSTRACT

Nowadays, many organizations analyze their data with the MapReduce paradigm, most of them using the popular Apache Hadoop framework. As the data size managed by MapReduce applications is steadily increasing, the need for improving the Hadoop performance also grows. Existing modifications of Hadoop (e.g., Mellanox Unstructured Data Accelerator) attempt to improve performance by changing some of its underlying subsystems. However, they are not always capable to cope with all its performance bottlenecks or they hinder its portability. Furthermore, new frameworks like Apache Spark or DataMPI can achieve good performance improvements, but they do not keep compatibility with existing MapReduce applications. This paper proposes Flame-MR, a new event-driven MapReduce architecture that increases Hadoop performance by avoiding memory copies and pipelining data movements, without modifying the source code of the applications. The performance evaluation on two representative systems (an HPC cluster and a public cloud platform) has shown experimental evidence of significant performance increases, reducing the execution time by up to 54% on the Amazon EC2 cloud.

1. Introduction

In the last several years, organizations have been using Big Data applications to extract valuable information from the huge data sets they manage. Many of these applications use the popular MapReduce programming model [1–3], which was first proposed by Google in [4]. The MapReduce paradigm allows high scalability and transparent parallelization by means of two explicit functions: Map and Reduce. Another important phase is data copy (also known as shuffle), which transfers intermediate data between mappers and reducers in a many-to-many fashion. Nowadays, Apache Hadoop [5], an open-source Java-based implementation of MapReduce, has become the de-facto standard framework for large-scale data processing.

As the capabilities demanded by Big Data applications increase continuously, the need for an efficient MapReduce framework has been growing, leading to a continuous refinement of the Hadoop implementation. However, some performance bottlenecks of Hadoop are caused by its underlying design. In order to overcome these limitations, previous works [6,7] have been mainly focused on modifying certain Hadoop subsystems (e.g., network communications, memory usage, I/O operations) to obtain better performance and scalability. These modifications keep compatibility with Hadoop APIs and so existing applications do not have to be rewritten. The main problem of this approach is that some design decisions remain in the core of the framework, affecting its performance. Another important issue is portability, as some of these modifications are written in other languages than Java (e.g., C, C++) or make certain assumptions (e.g., memory hierarchy features) about the system where they are being run. Furthermore, other options [8] involve replacing Hadoop with another framework, which implies changes in the source code of the applications. This is not always feasible for organizations that already have

* Corresponding author. Tel.: +34 881 011 212; fax: +34 981 167 160.
E-mail addresses: jorge.veiga@udc.es (J. Veiga), rreye@udc.es (R.R. Expósito), taboada@udc.es (G.L. Taboada), juan@udc.es (J. Touriño).

http://dx.doi.org/10.1016/j.future.2016.06.006
0167-739X/

many MapReduce applications in production or do not have access to their source code.

This paper introduces Flame-MR, a new event-driven design and implementation of the MapReduce model that enhances Hadoop in terms of memory efficiency, overlapping of the data flow between phases and leveraging of the computing resources. Furthermore, it ensures portability and full compatibility with existing applications. The main goal of Flame-MR is to provide the organizations a zero-effort way to decrease the execution time of their MapReduce workloads without having to modify or recompile their source code.

The rest of this paper is organized as follows: Section 2 introduces the related work. Section 3 describes the overall design and architecture of Flame-MR. Section 4 analyzes the performance evaluation of Flame-MR using several representative MapReduce workloads. Finally, Section 5 summarizes our concluding remarks and proposes future work.

2. Related work

The broad adoption of the Apache 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) [9] and RDMA-Hadoop [10], which adapt Hadoop to High-Performance Computing (HPC) resources, such as Remote Direct Memory Access (RDMA) interconnects like InfiniBand (IB). 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 [7]. On the other hand, RDMA-Hadoop redesigns the network communications to take full advantage of RDMA interconnects, while performing data prefetching and caching mechanisms [6]. 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 [11], 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 [12]. 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) [13], which uses the X10 programming language [14] to implement an in-memory 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.

The performance bottlenecks of Hadoop have caused the emergence of new frameworks that fully replace the Hadoop implementation. One of them is DataMPI [8], which makes use of the Message-Passing Interface (MPI) [15] to leverage the high-performance interconnects that are usually available in HPC systems. MapReduce Implementation Adapted for HPC Environments (MARIANE) [16] is designed to take advantage of the General Parallel Filesystem (GPFS), which is also commonly found in HPC systems. Other solutions, like Spark [17] and Flink [18], 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 frameworks 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.

Our previous work in [19] evaluated multiple MapReduce frameworks on an HPC cluster, providing some insights into their performance that have been used as a basis for developing Flame-MR. Unlike other frameworks, Flame-MR 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. Flame-MR design

This section presents the overall design of Flame-MR. First, the main characteristics of its internal architecture are discussed in Section 3.1. Second, Section 3.2 describes in more detail the different phases of the MapReduce data processing pipeline in this architecture.

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 Yet Another Resource Negotiator (YARN), which is its resource management layer, and uses the Hadoop Distributed File System (HDFS) [20] 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, merge, reduce and output. The input phase reads the input data set 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 data set 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 are executed within the same Java process (from now on, Worker). For a given data input, each operation performs some of the explained phases depending on its operation type (map, merge or reduce), as will be described in Section 3.2.

Fig. 1 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 Hadoop 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