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, España

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.

Keywords: Big Data, MapReduce, Hadoop, Event-Driven Architecture, Cloud Computing

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, 2, 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 per-

*Corresponding author: Tel: +34 881 011 212, Fax: +34 981 167 160

Email addresses: jorge.veiga@udc.es (Jorge Veiga), rreye@udc.es (Roberto R. Expósito), taboada@udc.es (Guillermo L. Taboada), juan@udc.es (Juan Touriño)

formance 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 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 dele-

gation 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.

Figure 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 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.

3.1.1. 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 (see Figure 1) 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

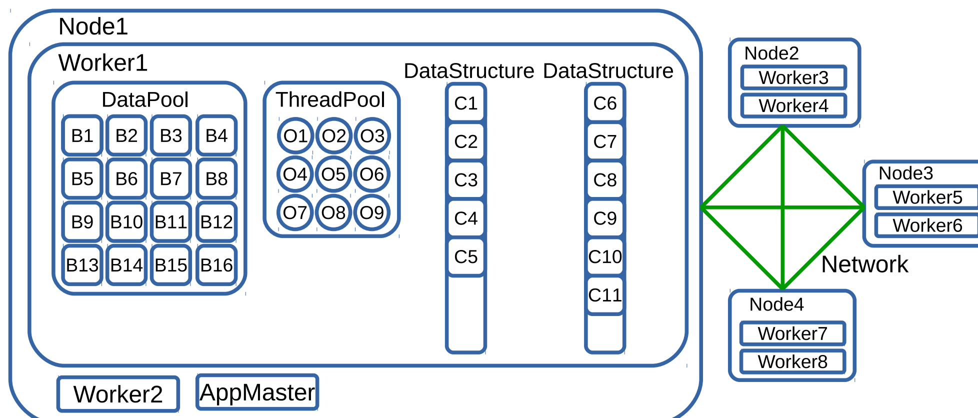


Figure 1: High-level architectural overview of Flame-MR
O: Operations B: Buffers C: Chunks

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 also relevant to optimize the I/O throughput.

3.1.2. Thread-based processing model

In Hadoop, each map/reduce task is executed in a YARN container, 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 utilization of memory 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 is given in Section 3.2.

3.1.3. Event-driven architecture

The relationship between the 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 DataStructures. 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, further shown in Figure 3). 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 of 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 software 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) [21]. 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.

3.1.4. Copy-avoidance mechanism

One of the main performance bottlenecks of Hadoop is the high number of extra 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, the equivalent Writable types in Flame-MR do not copy data fields from the input data 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 to 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

```java
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

```java
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 2: Code examples for WordCount map and reduce functions in Flame-MR and Hadoop

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.

3.1.5. Hadoop-like map and reduce functions

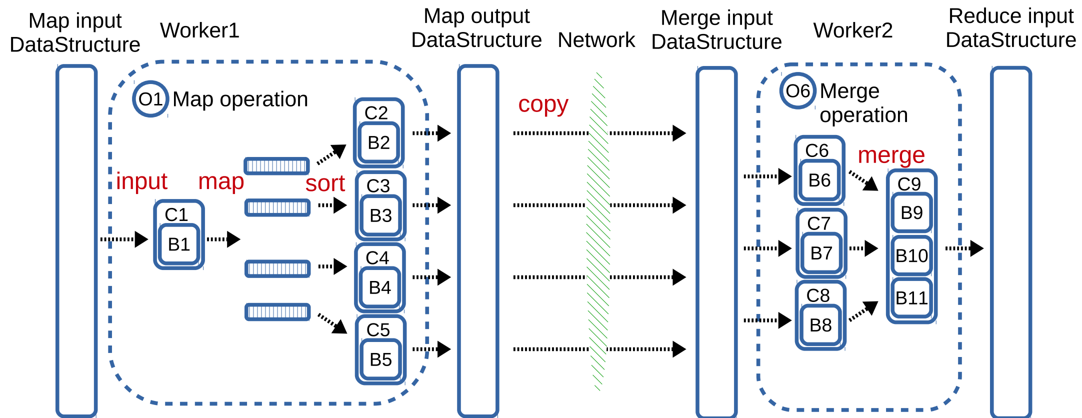
Existing Hadoop applications can be executed without any source code modification, as Flame-MR implements the Hadoop APIs. An example of the source code defined by the user is presented in Figure 2, which shows the map and reduce functions (Figures 2a and 2b, respectively) for the WordCount program. 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.

3.2. MapReduce operations

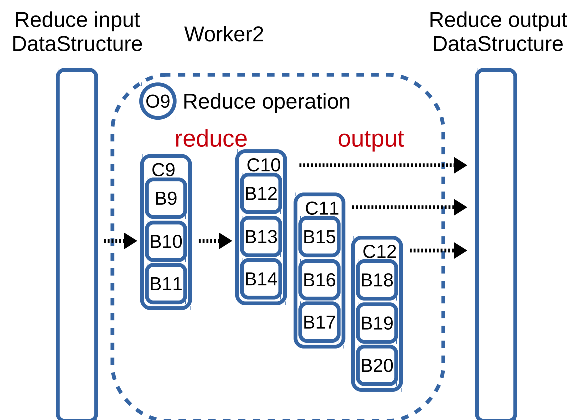
As mentioned before, the MapReduce data processing in Flame-MR is divided into three operation types: map, merge and reduce. These operations can be observed in Figure 3, which shows an overview of the whole data processing pipeline. More details about the computation that these operations perform and their specific optimizations are provided next.

3.2.1. Map operations

At the beginning of the job execution, the AppMaster (see Figure 1) divides the input data set 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 3a. 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 chunk resides in an in-memory



(a) MapReduce data flow of map and merge operations



(b) MapReduce data flow of reduce operations

Figure 3: Overview of the MapReduce workflow in Flame-MR
O: Operations B: Buffers C: Chunks

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 its resources

among the different partitions.

In-memory object sort. Unlike Hadoop, the 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 to avoid many memory copies during the sort phase. Furthermore, the copy-avoidance mechanism explained in Section 3.1.4 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 can not 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 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.

3.2.2. 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 3a. 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. Next, the operation merges these chunks until

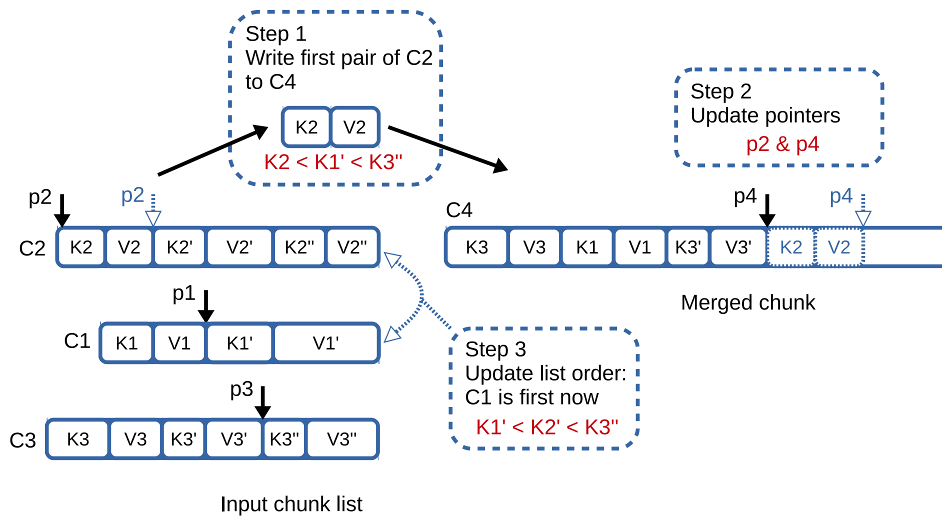


Figure 4: k-way merge (k=4)
C: Chunks K: Keys V: Values p: pointers

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.

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 which 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 4, 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 Java garbage collector.

3.2.3. Reduce operations

Reduce operations perform two phases: reduce and output. Their data flow is shown in Figure 3b. 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.

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.

4. Performance evaluation

This section presents the main results of the performance evaluation of Flame-MR compared with representative Hadoop-based MapReduce frameworks. First, Section 4.1 describes the benchmarks used in the evaluation and Section 4.2 the experimental configuration. Next, performance results are analyzed in Section 4.3.

4.1. Benchmarks

The main goal of this section is to evaluate the performance and scalability of Flame-MR and Hadoop-based MapReduce frameworks without changing the source code of the applications. To do so, the experiments have used several representative benchmarks, which are shown in Table 1. These benchmarks can be classified in two categories, micro-benchmarks and application benchmarks, depending on the nature of the computation they perform.

Micro-benchmarks. These benchmarks are used to obtain performance measurements about specific resources of a system, such as CPU, network or disk, by performing a simple computation which is mainly bound to one of them. The micro-benchmarks used in the experiments (Sort and Grep) are classic MapReduce workloads that can be found in any Hadoop distribution. Sort is mainly network and disk I/O bound, whereas Grep is CPU bound. The input data set of both benchmarks has been generated using the RandomTextWriter tool, also included in Hadoop.

Table 1: Benchmarks used in the experimental evaluation

Micro-benchmarks	
Sort	Sorts an input text data set
Grep	Extracts matching strings from text files and counts how many times they occurred
Application benchmarks	
PageRank	Ranks websites by counting the number and quality of the links to each one. Developed by Google, it is used to obtain Google search results
Connected Components	Explores a graph to determine its subnets

Application benchmarks. The performance of more complex scenarios has been evaluated by using the application benchmarks shown in Table 1, PageRank and Connected Components. These workloads are real-world MapReduce applications categorized as iterative algorithms, which perform several MapReduce jobs to compute the final result. These workloads have been taken from Pegasus 2.0 [22], a graph mining system built on top of Hadoop. The input data set of both applications has been generated by the Kronecker graph generator included in the BigDataBench suite [23].

4.2. Experimental configuration

The experiments have been carried out on two different systems: an HPC cluster and a public cloud infrastructure. The former is DAS-4 [24], a multi-core cluster with 8 cores and 24 GB of memory per node, interconnected via InfiniBand (IB) and Gigabit Ethernet (GbE). The latter is the Amazon Elastic Compute Cloud (EC2) [25], one of the most used and largest Infrastructure as a Service (IaaS) platforms both for HPC [26] and Big Data applications [27]. Amazon EC2 is offered by Amazon Web Services (AWS) and provides on-demand computing instances of different types using the pay-per-use model. Our experiments on EC2 have used the m3.2xlarge instance type, which provides 8 cores and 30 GB of memory per instance, interconnected via GbE. The instances are allocated in the standard AWS region (US East, North Virginia).

Table 2 shows the main hardware and software features of both systems. The experiments on DAS-4 have assessed the performance and scalability of the evaluated frameworks using 9, 17, 25 and 33 nodes. Each cluster size n can be understood as 1 master and $n - 1$ slave nodes. The main motivation of using Amazon EC2 is to obtain performance measurements that represent a real scenario in a public cloud platform. Hence, the experiments on EC2 have used the maximum cluster size that has been considered, 33 nodes, as a means of making full use of the paid resources.

Table 3 presents the MapReduce frameworks that have been evaluated. Besides comparing Flame-MR with standard Hadoop on both systems, the performance of certain flavors of Hadoop has also been considered. As mentioned in Section 2, these frameworks aim to improve the performance of Hadoop by adapting some of its subsystems to the computing environment where they are being run.

In the case of the DAS-4 cluster, Mellanox UDA [9] (Hadoop-UDA from now on), an HPC-oriented plugin for

Table 2: Hardware and software configuration

Hardware configuration	DAS-4 node	EC2 instance
CPU	2 × Intel Xeon E5620 Westmere	Intel Xeon E5-2670 v2 Ivy Bridge
CPU Speed/Turbo	2.4 GHz/2.66 GHz	2.5 GHz/3.3 GHz
#Cores	8	8
Memory	24 GB DDR3	30 GB DDR3
Disk	2 × HDD 1TB	2 × SSD 80 GB
Network	IB (40 Gbps) & GbE	GbE
Software configuration		
OS version	CentOS release 6.6	Amazon Linux AMI 2015.09
Kernel	2.6.32-358.18.1.el6.x86_64	4.1.10-17.31.amzn1.x86_64
Java	Oracle JDK 1.8.0_25	OpenJDK 1.7.0_91

Table 3: Evaluated MapReduce frameworks

Name	Version	Release Date	Network
DAS-4			
Hadoop	2.7.1	06/07/2015	IB (IPoIB)
Hadoop-UDA	3.3.2[a]	24/11/2013	IB (RDMA & IPoIB)[b]
Flame-MR	0.7.0	16/12/2015	IB (IPoIB)
EC2			
Hadoop	2.7.1	06/07/2015	GbE
EMR	4.2.0[c]	18/11/2015	GbE
Flame-MR	0.7.0	16/12/2015	GbE

[a] UDA was configured over Hadoop 2.7.1
[b] UDA uses RDMA communications in the copy phase and IPoIB for HDFS
[c] EMR 4.2.0 is based on Hadoop 2.6.0

Hadoop, has been selected. Although the evaluated version (3.3.2) is not the last update of this plugin, it is able to execute properly all the benchmarks used and has proven to obtain the best performance results. Besides Hadoop-UDA, RDMA-Hadoop [6] has also been assessed. However, as its results did not differ significantly from those of Hadoop-UDA, they have not been included in the graphs for clarity purposes, although further analysis and results of RDMA-Hadoop can be found in our previous work [19]. The network interface of the frameworks has been configured to use IP over InfiniBand (IPoIB), which provides higher bandwidth and lower latency than GbE. Hadoop-UDA also accelerates the copy phase by means of an RDMA protocol.

In the case of the cloud infrastructure, Amazon Elastic MapReduce (EMR) [28] has been evaluated. EMR is a cloud service that allows to automatically deploy a Hadoop cluster on Amazon EC2, tuning its configuration parameters and adding some improvements to make it work efficiently on this platform. The network interface used by all the frameworks was GbE, which is the only option available in this scenario.

Table 4 shows the size of the input data sets for the selected benchmarks, which have been configured in order

Table 4: Benchmark input sizes

Benchmark	DAS-4	EC2
Sort	50 GB	100 GB
Grep	10 GB	10 GB
PageRank & Connected Components	4M vertices	17M vertices

Table 5: Framework configuration

(a) Hadoop (also Hadoop-UDA for DAS-4)

Parameter	DAS-4	EC2
Mappers per node	4	4
Reducers per node	4	4
Mapper/Reducer heap size	2.3 GB	2.8 GB
HDFS block size	128 MB	128 MB
Replication factor	3	3

(b) EMR

Parameter	EC2
Mappers per node	12
Reducers per node	4
Mapper heap size	1.4 GB
Reducer heap size	2.9 GB
HDFS block size	128 MB
Replication factor	3

(c) Flame-MR

Parameter	DAS-4	EC2
Workers per node	2	2
ThreadPool size[a]	4	4
Worker heap size	9.7 GB	11 GB
DataPool size[a]	5 GB	6.4 GB
Data buffer size[b]	512 KB	1 MB
HDFS block size	128 MB	128 MB
Replication factor	3	3

[a] See Figure 1
[b] Size of each buffer Bx of the DataPool

to obtain a reasonable workload according to the capability of the systems and to keep the execution times into a reasonable range, especially important for the cloud experiments due to budget constraints. Moreover, the frameworks have been carefully configured according to their corresponding user guides and the characteristics of the systems (e.g., number of CPU cores, memory size). Table 5 shows the most relevant configuration parameters that have been used. In the case of Hadoop and Hadoop-UDA (see Table 5a) as well as Flame-MR (Table 5c), these parameters have been tuned in order to obtain an optimal configuration, ensuring a fair comparison. In the case of EMR (see Table 5b), these parameters are already tuned by default for the EMR Hadoop distribution and the m3.2xlarge instance type. Further information is available in the EMR release guide for Hadoop [29].

4.3. Performance results

This section presents the main results from the performance evaluation of the selected MapReduce benchmarks: Sort, Grep, PageRank and Connected Components. All Hadoop frameworks and Flame-MR have used the same source

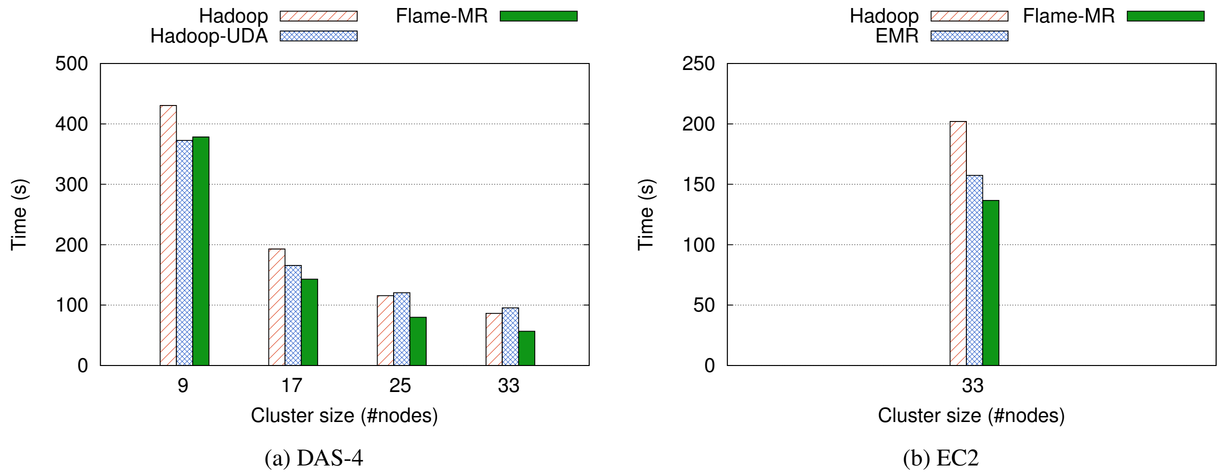
code of the benchmarks to carry out the evaluation. Furthermore, the MapReduce Evaluator tool (MREv, available at `http://mrev.des.udc.es`) [30] has been used in order to automate the experiments, including the configuration of the frameworks, the generation of the input data sets and the collection of results. The graphs in this section show the mean value from a minimum of 10 measurements for each experiment, although the observed standard deviations were not significant.

4.3.1. Sort

Figure 5 shows the execution times of the Sort benchmark. On DAS-4 (see Figure 5a), Flame-MR performance is better than Hadoop for all cluster sizes. Flame-MR also outperforms Hadoop-UDA except for 9 nodes, in which case the performance is roughly the same. With this cluster size, the available memory of the nodes gets full due to the I/O bound behavior of Sort, as the map operations copy all their input to the map output. This involves a high number of spilling operations to local disks, with an average disk usage of approximately 66% for Hadoop and Flame-MR, and 88% for Hadoop-UDA. Hence, the disk bandwidth limits the performance of the workloads. As the number of nodes increases, the frameworks are not so constrained by the available memory, reaching average disk usage values of 47%, 49% and 45% for Hadoop, Hadoop-UDA and Flame-MR, respectively, with 33 nodes. Note that these measurements include the writing of the final output to HDFS. As a consequence of the increase in the available memory, Flame-MR achieves higher scalability and the performance gap becomes wider, obtaining up to 34% improvement compared with Hadoop when using 33 nodes (40% with respect to Hadoop-UDA in the same case). A similar analysis can be drawn from the results on Amazon EC2 (see Figure 5b). Flame-MR outperforms Hadoop and EMR on the cloud, obtaining a performance improvement of 32% and 13%, respectively. Note that EMR is a framework optimized for the EC2 platform and thus the performance gain of Flame-MR is lower.

4.3.2. Grep

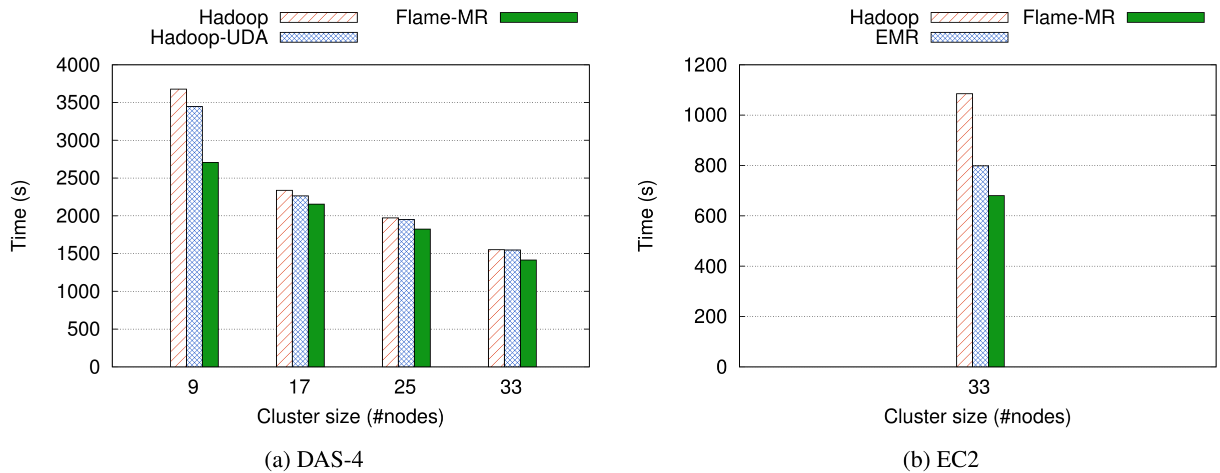
The execution times for the Grep benchmark can be seen in Figure 6. The results on DAS-4 (Figure 6a) show that Flame-MR obtains again the best performance in all cases. However, the performance improvement becomes lower as the number of nodes increases. This is caused by the nature of Grep, which is mainly CPU bound. Most part of the computation processes the user-defined map function, analyzing the regular expression and incurring high CPU usage. Using a higher number of nodes allows the frameworks to use more CPU cores and thus reduce equally their execution time. This situation causes the performance improvement of Flame-MR to be 26% with 9 nodes (compared with Hadoop), but limited to 9% with 33 nodes. Moreover, the execution with 9 nodes shows a reduction of 41% in the memory usage of Flame-MR with respect to Hadoop, which confirms that the design of Flame-MR improves memory efficiency. Regarding Amazon EC2 (see Figure 6b), Flame-MR is clearly the best performer, obtaining a performance improvement compared with Hadoop and EMR of 37% and 15%, respectively. Note that these improvements are better than on DAS-4 when using 33 nodes. Comparing both systems, m3.2xlarge instances on EC2 are based on Ivy Bridge (see Table 2), a much more modern CPU microarchitecture than Westmere, the technology behind the



(a) DAS-4

(b) EC2

Figure 5: Sort execution times



(a) DAS-4

(b) EC2

Figure 6: Grep execution times

DAS-4 nodes, which results in more performance per core. Moreover, the disk technology on DAS-4 nodes is based on mechanical Hard Disk Drives (HDD), while EC2 instances provide Solid State Drives (SSD), which obtain higher disk throughput than HDD. However, the disk technology has a negligible impact on the performance of Grep, as it is mainly CPU bound and the input data size is 10GB, much lower than in Sort. Therefore, while the execution time of Grep is more constrained by the CPU speed on DAS-4, the memory management acquires more importance on EC2, thus allowing Flame-MR to widen the performance gap on the cloud scenario. In this case, the reduction in memory usage of Flame-MR with respect to Hadoop is 73%, which is caused by the design features previously explained, especially the copy-avoidance mechanism (see Section 3.1.4) and the binary comparisons (Section 3.2.2).

4.3.3. PageRank

The performance results for PageRank are shown in Figure 7. PageRank takes 30 iterations (i.e., MapReduce jobs) to compute its final result, thus the potential performance gain for each iteration is accumulated to the overall execution time. As can be seen in Figure 7a, the scalability of PageRank on DAS-4 is limited by the iterative nature of the workload, as a considerable part of the execution is consumed by the launching/finalization overhead of the MapReduce jobs. This overhead is lower for Flame-MR than for Hadoop. The main reason is that Flame-MR uses a thread-based task management (see Section 3.1.2), whereas Hadoop allocates a YARN container per each new map/reduce task. Therefore, the event-driven architecture of Flame-MR (explained in Section 3.1.3) also allows to improve the task management overhead of Hadoop, being clearly more suited for iterative workloads. This behavior is also shown on EC2 (see Figure 7b). It can be observed that the performance improvement of Flame-MR with respect to Hadoop-based frameworks is roughly 50% on both systems.

4.3.4. Connected Components

Figure 8 presents the performance results of Connected Components on DAS-4 (Figure 8a) and EC2 (Figure 8b). The analysis that can be extracted from these results is the same as the PageRank benchmark. It can be seen that the scalability on both systems keeps being limited by the iterative nature of this workload. Consequently, the performance improvement of Flame-MR compared with Hadoop is 50% and 54% on DAS-4 and EC2, respectively.

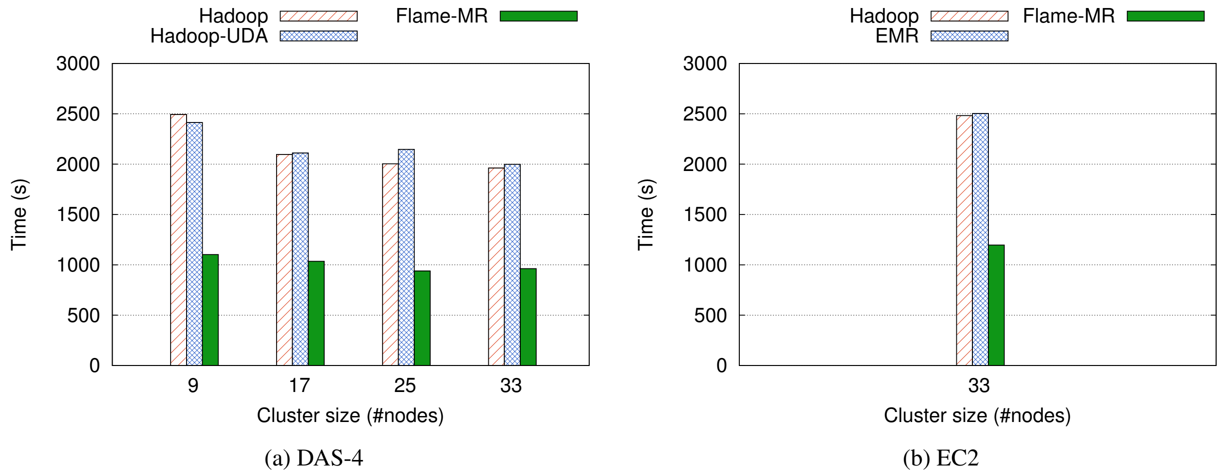
Both PageRank and Connected Components show little difference between the performance of Hadoop and Hadoop-UDA on DAS-4 and between Hadoop and EMR on EC2. For iterative workloads, the optimizations included in both flavors of Hadoop do not seem to improve the performance. In this case, Flame-MR provides greater benefit in terms of execution time.

5. Conclusions and future work

Apache 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 paper has presented Flame-MR, a new MapReduce framework that improves the performance of Apache Hadoop without modifying the source code of the applications. Flame-MR transparently replaces the inner 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. Furthermore, it pipelines the output of map and reduce phases to decrease the disk usage and improve the overlapping of data processing with disk and network operations.

In order to evaluate the benefits of Flame-MR[1], its performance has been analyzed comparatively with representative Hadoop-based MapReduce frameworks on an HPC cluster and on an IaaS cloud platform (Amazon EC2). The

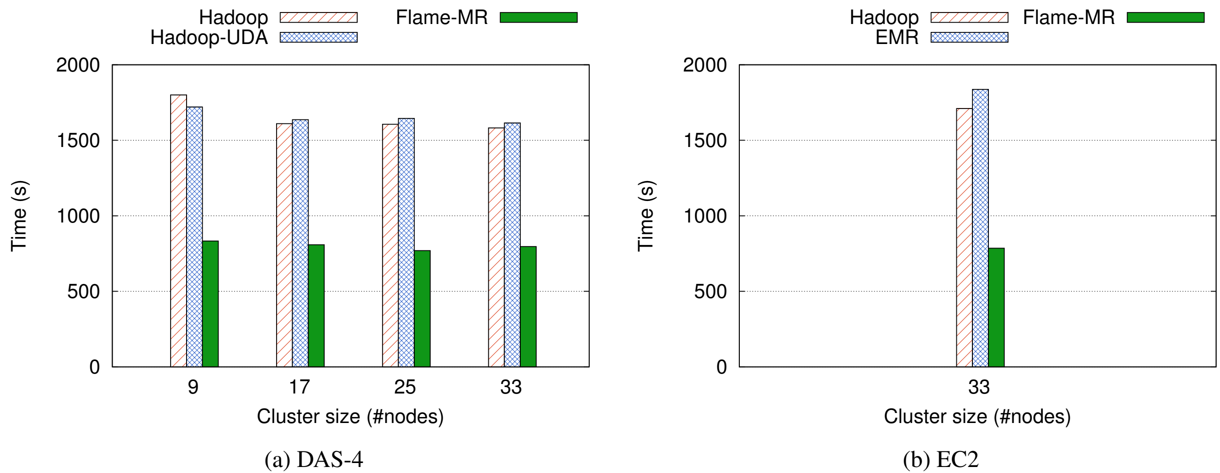
[1] A prototype version of Flame-MR for testing purposes is available at http://flamemr.des.udc.es



(a) DAS-4

(b) EC2

Figure 7: PageRank execution times



(a) DAS-4

(b) EC2

Figure 8: Connected Components execution times

analysis of the results has revealed that Flame-MR can improve the performance and scalability of I/O bound workloads (e.g., Sort), reducing the execution times by up to 34% (compared with Hadoop). Moreover, the event-driven architecture of Flame-MR allows to improve system resource utilization (CPU, memory, network and disk), providing significant performance improvements of up to 54% for iterative workloads (Connected Components).

In the near future, we plan to further improve the Flame-MR performance by enhancing the memory management of data buffers, reducing unnecessary spilling operations to disk and minimizing the impact of the Java garbage collector by using off-heap memory. We also plan to redesign the way key-value pairs are stored during the copy and the merge phases, in order to reduce the amount of data sent through the network. Additionally, we aim to improve the fault tolerance of Flame-MR without reducing its overall performance.

Acknowledgments

This work was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the European Union (Project TIN2013-42148-P), by the FPU Program of the Ministry of Education (grant FPU14/02805), and by an Amazon Web Services (AWS) LLC research grant. We thankfully acknowledge the Advanced School for Computing and Imaging (ASCI) and the Vrije University Amsterdam for providing access to the DAS-4 cluster.

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

[2] J. Lin, A. Kolcz, Large-scale machine learning at Twitter, in: Proceedings of the 2012 International Conference on Management of Data (SIGMOD'12), Scottsdale, AZ, USA, 2012, pp. 793–804.

[3] S. Jeon, B. Hong, Monte Carlo simulation-based traffic speed forecasting using historical Big Data, Future Generation Computer Systems (In press). doi:10.1016/j.future.2015.11.022.

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

[5] Apache Hadoop, http://hadoop.apache.org/, [Last visited: April 2016].

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

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

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

[9] Mellanox Technologies: Hadoop, http://www.mellanox.com/page/hadoop, [Last visited: April 2016].

[10] High-Performance Big Data (HiBD) project, http://hibd.cse.ohio-state.edu/, [Last visited: April 2016].

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

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

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

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

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

[16] Z. Fadika, E. Dede, M. Govindaraju, L. Ramakrishnan, MARIANE: using MApReduce in HPC environments, Future Generation Computer Systems 36 (2014) 379–388.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2nd USENIX Conference on Hot topics in Cloud Computing (HotCloud'10), Boston, MA, USA, 2010, pp. 10:1–10:7.

[18] Apache Flink: scalable batch and stream data processing, http://flink.apache.org/, [Last visited: April 2016].

[19] J. Veiga, R. R. Expósito, G. L. Taboada, J. Touriño, Analysis and evaluation of MapReduce solutions on an HPC cluster, Computers & Electrical Engineering (In press). doi:10.1016/j.compeleceng.2015.11.021.

[20] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010), Incline Village, NV, USA, 2010, pp. 1–10.

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

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

[23] 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, B. Qiu, BigDataBench: a Big Data benchmark suite from Internet services, in: Proceedings of the 20th IEEE International Symposium on High-Performance Computer Architecture (HPCA'14), Orlando, FL, USA, 2014, pp. 488–499.

[24] Advanced School for Computing and Imaging (ASCI): Distributed ASCI Supercomputer, version 4 (DAS-4), http://www.cs.vu.nl/das4/, [Last visited: April 2016].

[25] Amazon Web Services Inc. Amazon Elastic Compute Cloud (Amazon EC2), https://aws.amazon.com/ec2/, [Last visited: April 2016].

[26] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, R. Doallo, Performance analysis of HPC applications in the cloud, Future Generation Computer Systems 29 (1) (2013) 218–229.

[27] T. Gunarathne, T.-L. Wu, J. Qiu, G. Fox, MapReduce in the Clouds for Science, in: Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10), Indianapolis, IN, USA, 2010, pp. 565–572.

[28] Amazon Web Services Inc. Amazon Elastic MapReduce (Amazon EMR), https://aws.amazon.com/emr/, [Last visited: April 2016].

[29] Amazon Web Services Inc. Amazon EMR: Apache Hadoop, http://docs.aws.amazon.com/ElasticMapReduce/latest/ReleaseGuide/emr-hadoop.html, [Last visited: April 2016].

[30] J. Veiga, R. R. Expósito, G. L. Taboada, J. Touriño, MREv: an automatic MapReduce Evaluation tool for Big Data workloads, in: Proceedings of the International Conference on Computational Science (ICCS'15), Reykjavík, Iceland, 2015, pp. 80–89.