Provisioning Input and Output Data Rates in Data Processing Frameworks

Nam H. Do · Tien Van Do · Lóránt Farkas ·
Csaba Rotter

Received: 20 June 2018 / Accepted: 1 January 2020 / Published online: 3 March 2020
© The Author(s) 2020

Abstract This paper is motivated by the need of deadline-bounded applications in live mobile network environments to obtain the guarantee and the appropriate share of an input and output (I/O) data rate. However, data processing frameworks only support the request of memory and the computing capacity at present. In this paper, we propose a solution that allows the control of disk I/O and network I/O for data processing applications in YARN and Mesos frameworks. Experimental results show that our tool can provision the I/O data rate sharing of competing data processing applications.

Keywords I/O enforcement · I/O data rate control · Cluster resource management · Hadoop YARN · Apache Mesos · HDFS

1 Introduction and Motivation

When a specific application submits a job, a data processing framework such as Apache Hadoop [1, 26], Hadoop YARN [25], Mesos [2], reserves and allocates necessary computing resources for the execution of the job. Many of the resource models include the amount of memory, the number of virtual CPU cores, but do not contain information on the disk I/O capability of the commodity servers and the network I/O rate. Since a computing cluster can be built up of heterogeneous hardware and software components, the I/O data rate perceived by applications is unpredictable due to the contention for the resources of the physical servers, and there is a need to monitor applications running on these platforms as well [12]. As shown in [11, 24], the I/O contention of applications leads to the degradation of quality of service.

Nowadays telecommunication operators often apply frameworks to regularly process big data sets with specific deadlines in their computing clusters. Therefore, the I/O data rate guarantee and the control of sharing the available data rate for applications are critical issues in the environment of telecommunication operators. When applications compete for a resource in the hardware and the network level, which is hidden from programmers (and therefore from applications), they may suffer the I/O performance degradation.

T. V. Do (✉)
Baoji University of Arts and Sciences, Shaanxi, China
e-mail: do@hit.bme.hu

N. H. Do · T. V. Do
Department of Networked Systems and Services, Budapest
University of Technology and Economics, Magyar tudósok
körútja 2., Budapest, Hungary

L. Farkas · C. Rotter
Nokia Bell Labs Hungary, Bókay János utca 36 - 42,
Budapest, Hungary

Motivated by the need, we design a complete solution that can be applied to provision the I/O data rate of applications in both the Mesos and YARN frameworks. Note that this is a result¹ that has gradually been improved over the years based on our previous experiences [10, 11, 24]. We demonstrate that the proposed functionalities can be integrated into two popular data processing frameworks such as Mesos and YARN to control the I/O data rates (disk I/O and network I/O) of applications, which may relieve the pain of service providers on the integration of schedulers to existing frameworks. Also, we show that our solution can be applied to provision the share of the I/O data rate of competing and deadline-bounded applications.

The rest of this paper is organised as follows. A summary of related works is provided in Section 2. The design and the implementation are described in Section 3. Experimental results are presented in Section 4. Section 5 concludes the paper.

2 Related Work

In this section, we provide a short overview of papers that dealt with the issue related to our work which has gradually been improved over years.

Kc and Freeh [15] proposed a feedback-based dynamic approach for controlling the parallelism of concurrent containers on the node level. This method takes into account the percentage of time spent by the CPU in user mode, a number of processes blocked on I/O data, and a total number of context switches of each worker node to suit an application and avoid bottlenecks for all types of MapReduce applications in a YARN cluster. The characteristics of MapReduce workloads (such as CPU-bound and I/O-bound) are considered in scheduling algorithms for improving the performance of the Hadoop cluster [17]. Ko et al. [16] addressed the overhead of I/O data block processing in virtualized Hadoop clusters and propose a large segment scheme for I/O ring (the structure between frontend and backend driver for transferring I/O requests) to improve the performance perceived by applications. Spivak and Nasonov [23] suggested a distributed cache to preload data before its computation in Hadoop systems. They demonstrated

that their method effectively reduces the execution time of MapReduce jobs, particularly when the operation time of I/O data is lower than the intensive phase of the CPU. Jung et al. [21] proposed to integrate Hadoop Distributed File System (HDFS) with a lightweight Adaptive I/O system that supports various I/O-based methods. The study shows that by choosing the optimal I/O method for a particular platform, the reduction of I/O time increases the system performance. Malensek et al. [19] studied a disk I/O contention issue between virtual machines and proposed a scheduler based on Linux tools to provide the share of a disk I/O. Enes et al. [13] investigated the resource contention and its effects on PaaS services. They proposed a Platform as a Service (PaaS) architecture where a disk I/O is offered as resources. Liao et al. [18] presented a new scheme of I/O scheduling to yield better I/O performance in the servers of distributed/parallel file systems. Amamou et al. [9] proposed a dynamic bandwidth allocator that limits the network bandwidth for virtual machines in a cloud environment. Kamal et al. [15] proposed a feedback-based dynamic approach for controlling the parallelism of concurrent containers on node level. Their proposal takes into account the percentage of time spent by the CPU in user mode, a number of processes blocked on I/O data, and a total number of context switches of each worker node to suit an application and avoid bottlenecks for all types of MapReduce applications in a YARN cluster.

The idea of co-locating different scheduler frameworks in the same data center has been in discussion for the benefit of the operators and service providers. For example, only HDFS read traffic shaping in a YARN cluster using Traffic Control (LTC) mechanism was proposed in [11], while the disk I/O problem for Hadoop MapReduce applications and Spark applications [3, 28] was investigated in [10, 24]. Xu and Zhao [27] presented an interposed big-data I/O scheduler to provide I/O performance differentiation for competing applications in a shared big-data system. It is worth mentioning that the work by Xu and Zhao [27] is close to our previous solution [11]. However, they did not present the detail of their solution. In this paper, we propose a tool with a set of functionality to monitor and control the competition of applications in production environments, which is the result of improving works presented in [10, 11, 24]. The main contribution of this paper is the

¹<https://github.com/dohona/hadoop>

systematical design of a tool based on exploring necessary functionality needed in both the Mesos and YARN frameworks. Furthermore, we show the provisioning capability of our tool in a real use case with competing and deadline-bounded applications.

3 Functionalities to Control the I/O Data Throughput of Applications

In this Section, we provide an overview of the design and implementation of our tool.

3.1 Design

Our aim is a set of functionalities that control and enforce the I/O data rate for applications in data processing frameworks. Following the ITU-T G.1000 [22] recommendation, a procedure to guarantee resources, therefore, should include:

- the specification of resource requirements of users and their applications,
- information about the shared compute cluster,
 - the maximum capacity of the resource (e.g. disk I/O capacity) of the cluster,
 - the amount of resources occupied by containers and applications,
- a unified resource management strategy,
 - a set of policies to schedule and allocate the resources for applications,

- a set of mechanisms to isolate a granted amount of the resource given to a container,
- a set of features and tools to enhance the ability to perform and automate operational tasks such as configuration, monitoring, auditing, etc.

As a consequence, we propose software components for I/O monitoring and enforcement in Fig. 1. *Controller* keeps information about the capability of physical servers and also maintains the status information of agents. Agents that run in physical servers must report the capacity of the new resource type of the corresponding worker node, monitor the amount of new resource type occupied by containers, and apply an isolation mechanism to enforce the resource usage of containers. The agent executed in each physical machine contains two components:

- (i) *ResourceMonitor* monitors the resource usage of the running containers and the worker machine.
- (ii) *IOController* performs resource enforcements.

The tool forms a layer between a big data processing framework and an underlying operating system. The tool maps the abstract resource quantities to the real capabilities of commodity servers and networks. The tool interacts with the underlying operating system to monitor and enforce the use of resources in the computing cluster and the data flow of HDFS. Furthermore, it provides monitoring data, such as the resource usage of the containers, Data Nodes, etc. as well as the free resources for the resource management

Fig. 1 Architectural design of the proposed solution

frameworks. For the coordination of the controller with agents and data frameworks (i.e., the exchange of resource reservation, allocation, monitoring and enforcement information), the following interfaces are defined.

- Interface **I1** is to configure the Controller and agents.
- Interface **I2** is to exchange data on I/O resource usage between our tool and data processing frameworks.
- Interface **I3** is to obtain information of containers and I/O data rate requirements.
- Interface **Ca1** is to exchange data between the agents and the Controller.

3.1.1 The ResourceMonitor Component

The software modules of the ResourceMonitor component are plotted in Fig. 2. The ResourceMonitor is responsible for

- registering and maintaining the information of containers requested by the computing frameworks,
- monitoring the resource usage of the containers that have been started,
- collecting all processes belonging to each container,
- observing all established HDFS connections and collecting host:port information of each container,
- forming and sending enforcement requests to IOController component in order to modify and perform an appropriate enforcement,
- unregistering and stopping the action that monitors the resource usage of a container.

When a new container is launched by YARN (or Mesos), NodeManager of YARN (or Mesos Slave) send the information through an interface to the *ResourceMonitor* component residing in each machine. A container registration message includes

- the unique string identifier (*containerId*) of a container,
- the identifier (*pid*) of the process spawned to handle the container, and
- the resource requirements (with the resource type, HDFS read limit rate, disk I/O read throttling rate).

The details of the registration process when a new container is launched by YARN or Mesos are presented in Sections 3.2.1 and 3.2.2.

The *ResourceMonitor* has the following main software modules.

- *ContainerManager* provides an interface for registering containers reported by the slaves such as YARN NodeManager or Mesos slave.
- *Proc Monitor* detects the establishment of TCP connections and collects the incoming HDFS connections of a given container in an accumulative way by subsequently walking the */proc/net* directory (containing networking parameters and statistics in a Linux machine) and parsing the output of Linux built-in *ss* utility – that displays socket statistics. *Proc Monitor* also monitors a container's resource usage, periodically collects the child processes of the reported process. Then, it either delivers this data to the *IOController* component or submits them to the Controller.

Fig. 2 The modules of the Resource Monitor

The cooperating frameworks report the establishment of the launched containers through an interface with container data (e.g. pid, resource requirements). After the registration of a container, the *Proc Monitor* periodically monitors the container to collect the list of processes belonging to the container as well as the list of its connections to HDFS, besides other resource usages. They are used to form enforcement requests that are sent to the adjacent *IOController* for enforcing the local disk I/O or for shaping the local HDFS read traffic. If an HDFS read connection involves a remote DataNode, then the appropriate enforcement request is submitted to the agent. It is worth mentioning that in the case of YARN MapReduce, one container is spawned to handle only one map/reduce task. However, in the case of Spark on Mesos, one container is spawned to handle a Spark executor, which in turn may run many tasks in parallel. Hence the requirement specifications (e.g. the HDFS read limit rate) should be determined based on the number of current tasks in the container and are adjusted during the lifetime of the container.

3.1.2 The IOController Component

The *IOController* components of the agent are illustrated in Fig. 3. The *IOController* performs an enforcement based on the information from the *ResourceMonitor* component and the *Controller*. The *IOController* component has the following modules.

- *Forwarder* receives and routes the enforcement requests to the appropriate handler modules.

- *NetworkTrafficController* enforces the data rate for TCP/IP connections between the DataNode and a particular container. TCP flows are grouped into classes specified by filters of various conditions (e.g., priority, interface, or originating program). TCP/IP traffics in one class equally share bandwidth. Classes are enforced by the class-based queueing scheduler –CBQ [6]. *LTCEventBuilder* build appropriate filters to classify data flows. Then, the information is passed to *TrafficControllDeviceExecutor* to enforce the I/O data rate on flows with the use of Linux *tc* program. Note that there may be more instances of *TrafficControllDeviceExecutor* in one machine. Each instance corresponds to one Network interface card (NIC) device. *TcFilterExecutor* is used to create and specify the class a specific flow belongs to, while *TcClassExecutor* is for giving the enforcement instruction for the CBQ scheduler. It is worth emphasizing that the control information of HDFS belongs into a class of the highest priority.
- *DiskIOController* performs a disk I/O enforcement for requested containers. *DiskIOController* has two submodules to perform a disk I/O enforcement.
 - *IOMapper* provides the mapping of abstract resource quantities in the enforcement request (used by a cluster resource management framework) to the real capabilities of commodity servers (used by the underlying operating systems).

Fig. 3 The modules of the IOController component

Fig. 4 Software modules in the proof-of-concept framework

- *Enforcer* interacts with *CGroups Block I/O Controller*² to control disk I/O activities based on the information from *IOMapper*.

The *IOController* is also responsible for cleaning the settings of the *CGroups Block I/O Controller* and LTC-based TCP/IP Controller when a container is removed. For example, when HDFS is enforced, the *ResourceMonitor* submits the container an empty list of connections. Then all LTC settings related to this container are deleted in each related *DataNode* machine. In the initialization phase, the *TrafficControlDeviceExecutor* collects all the old LTC settings. These data are then merged with the up-to-date information to generate the new LTC settings for the enforcement. The old settings are deleted, and the new LTC settings are applied.

3.2 Implementation

The software components (Fig. 4) have been implemented. The Controller applies ZooKeeper to store data and communicate with the agents, and to route enforcement requests to the corresponding agents. The data structure used by the Controller was presented in [11]. Agents use the Linux built-in utilities to monitor and enforce I/O data rates.

The interface **I3** was implemented as a file-based plugin as follows.

- In a machine, an agent is configured to monitor a specific folder periodically.
- External frameworks' components (e.g. a Mesos slave, a YARN NodeManager) can report containers

and submit enforcements requests by creating or modifying flag files in a given folder.

- Each of the flag files contains all information and enforcement requests for an individual container. The enforcement requests can be modified during the lifetime of the container by changing the content of the corresponding flag file.
- When the agent detects changes in the folder (e.g. new files are created, or some files are modified, deleted), it parses the content of the files in the folder and uses them to synchronize its persistent data as well as to construct and submit enforcement requests to the Controller.

3.2.1 Cooperation with YARN

The following steps are required to cooperate with YARN.

- The *NodeManager* communicates with the agent to add, modify or remove containers.
- After receiving the resource monitoring and enforcement requests from the *NodeManager* (which shall contain the container ID, the enforcement type and the resource requirements for each container), the agent starts the procedures to monitor and enforce resources for the requested containers. To perform resource enforcement, the agents share data through the Controller.
- The *NodeManager* may modify existing enforcement requests during the operation.

When a YARN client submits a job to the *ResourceManager*, the submission of the job contains the resource requirements for a container that hosts the *Application Master (AM)* of the client [20, 26]. The *AM* is responsible for requesting resources for a container or containers to run application tasks. An instance of *Resource* class conveys the resource

²a Linux kernel feature to control the resource usage of the group of processes [4]

The execution of the MapReduce WordCount (that reads files stored in HDFS and count the occurrence of words) with the proposed components is depicted in Fig. 5.

- N) Based on the list of tasks and the I/O data rate requirements (per task), a NodeManager (NM) calculates the total amount of resources for a given container, and reports a resource enforcement request to the client.
- E) The IOController then translates the request into appropriate enforcement settings and applies LTC rules to shape the requested HDFS downstream traffic. The enforcement is performed during the lifetime of the container.
- M) The *ResourceMonitor* monitors the HDFS read connections of the given container and may submit the collected connection information to the Controller for a remote *IOController*.

The following aspects need to be considered to cooperate with Mesos:

-
- The diagram illustrates the Hadoop MapReduce architecture, showing the interactions between various components across different machines:
- Machine (Top Left):** Contains a **Name Node**.
 - Machine (Top Center):** Contains a **WordCount Job Object**.
 - Machine (Top Right):** Contains a **Resource Manager**.
 - Worker Machine (Left):**
 - Node Manager:** Interacts with the Name Node (Get input split), WordCount Job Object (S: start containers), and Resource Manager (S: launch & enforce).
 - DataNode:** Contains **blk1** and **blk2**. It interacts with the Node Manager (S: launch & enforce), YarnChild (get job resources), and Map Task 1 (execute).
 - YarnChild:** Interacts with the Node Manager (S: launch & enforce) and Map Task 1 (execute).
 - Map Task 1:** Interacts with the YarnChild (execute) and Map Task 2 (execute).
 - Map Task 2:** Interacts with the YarnChild (execute) and Map Task 1 (execute).
 - JVM:** Contains the YarnChild and Map Task 1.
 - JVM:** Contains the YarnChild and Map Task 2.
 - Worker Machine (Right):**
 - Node Manager:** Interacts with the Resource Manager (S: launch & enforce) and Data Node (S: start container).
 - Data Node:** Interacts with the Node Manager (S: start container), YarnChild (get job resources), and Reduce Task (write output).
 - YarnChild:** Interacts with the Node Manager (S: start container) and Reduce Task (write output).
 - Reduce Task:** Interacts with the YarnChild (write output) and Data Node (write output).
 - JVM:** Contains the YarnChild and Reduce Task.
 - Machine (Bottom Center):** Contains a **Controller**.
 - Machine (Bottom Left):** Contains an **IO Controller** and a **Resource Monitor**.
 - Machine (Bottom Right):** Contains an **IO Controller** and a **Resource Monitor**.
 - Worker Machine (Left):**
 - IO Controller:** Interacts with the Resource Monitor (M: start monitor container) and the Controller (M: submit data).
 - Resource Monitor:** Interacts with the IO Controller (M: start monitor container) and the Controller (M: submit data).
 - Worker Machine (Right):**
 - IO Controller:** Interacts with the Resource Monitor (M: start monitor container) and the Controller (M: submit data).
 - Resource Monitor:** Interacts with the IO Controller (M: start monitor container) and the Controller (M: submit data).
 - Machine (Bottom Center):**
 - Controller:** Interacts with the IO Controller and Resource Monitor (M: submit data) on both Worker Machines, and the Resource Manager (E: enforce I/O).
 - Resource Monitor:** Interacts with the IO Controller (M: submit data) and the Controller (E: enforce I/O).
 - IO Controller:** Interacts with the Resource Monitor (M: submit data) and the Controller (E: enforce I/O).
 - Machine (Bottom Left):** Contains a **Resource Monitor** and an **IO Controller**.
 - Machine (Bottom Right):** Contains a **Resource Monitor** and an **IO Controller**.
 - Machine (Bottom Center):** Contains a **Controller**.
 - Machine (Bottom Left):** Contains an **IO Controller** and a **Resource Monitor**.
 - Machine (Bottom Right):** Contains an **IO Controller** and a **Resource Monitor**.
 - Machine (Bottom Center):** Contains a **Controller**.
 - Machine (Bottom Left):** Contains an **IO Controller** and a **Resource Monitor**.
 - Machine (Bottom Right):** Contains an **IO Controller** and a **Resource Monitor**.
 - Machine (Bottom Center):** Contains a **Controller**.

 Springer

enforcement type and resource requirements for each container) the agent starts procedures to monitor and enforce an I/O rate for the requested containers. To perform resource enforcement, the agent may share data with other agents through the Controller.

- Mesos slaves may request the monitored data from the agent to report its data usage to the Mesos master.

In Fig. 6 the execution flow of the Spark WordCount example with the proposed components is depicted.

- S) A Mesos slave retrieves information from the *ResourceMonitor* and reports the amount of usable resource (e.g. I/O data rate) to the Mesos master, which in turn advertises it to a specific Spark driver. The Spark driver must specify I/O data rate requirements when a resource offer is accepted.

To shape the HDFS downstream traffic of containers, Mesos slaves must report the HDFS read limit bandwidth to the Mesos master. The application driver should specify its requirement for the resource by consuming the offers from the Mesos master. The resource model of Mesos assures that the requested HDFS read limit of a container is passed to a corresponding Mesos

slave, which in turn launches a container and assigned tasks.

- N) Based on the list of tasks and the I/O data rate requirements (per task), the Mesos slave can calculate the total amount of the resource for the given container, then it can notify the agent about the resource enforcement request. As a part of the resource enforcement procedure, the Mesos slave calculates the amount of HDFS read limit rates for the given container, based on its running tasks, and submits the request to the adjacent agent. For example, in case of the mentioned file-based plugin it must export the container data into a flag file, the name of which is in the *ContainerId* string of the message, in a pre-configured folder processed by the adjacent agent.
- M) The *ResourceMonitor* monitors the container, then forms and sends enforcement requests to the *IOController*. It may share data with a remote *IOController* via the Controller.
- E) The *IOController* in turn translates the request into the appropriate enforcement settings, and the enforcement is performed during the lifetime of a given executor. The *ResourceMonitor* monitors the HDFS read connections of the given container and sends the collected connection data to the *IOController* through the Controller.

Fig. 6 Modified execution flow of Spark WordCount example with the proposed approach

The *IOController* component then applies LTC rules to shape the requested HDFS downstream traffic. The setting for the HDFS read limit of the container can be adjusted during the container's lifetime based on its current tasks.

4 Experimental Results with Controlling I/O Activities

In this section, we present numerical results to show that I/O activities can be controlled with the proposed framework.

4.1 Testbed and Performance metrics

Experiments are performed in a shared cluster that consists of the following elements:

- Intel® Core™ i5-4670 CPU @ 3.40GHz
- 16GB Kingston HyperX Black DDR3 1600MHz RAM
- Asus H87-PLUS Motherboard with Intel® H87 chipset and SATA 6Gb/s Ports
- 1TB WD Black WD1003FZEX 7200 RPM hard drives with SATA 6Gb/s interface and data transfer rate up to 150 MB/s

Ubuntu Server 16.04.3 LTS 64 bit, Hadoop 2.7.5, Mesos 1.5.0, and Spark 2.3.0 were installed in our cluster. Note that the HDFS files are stored in a separate hard drive in DataNode machines.

Sequential reading or writing activities on a large number of disk blocks of an application may seize the whole I/O capacity of a disk and cause bottlenecks for other applications that access the same disk. Such cases can frequently occur in live mobile network environments when data are uploaded to HDFS or reorganized in HDFS. To emulate such cases, we use some I/O-greedy applications.

- (i) HDFS-reader and HDFS-writer are simple applications for reading and writing HDFS data. HDFS-writer randomly generates and repeatedly writes a text paragraph into an output file in HDFS until the given file-size is reached, while HDFS-reader reads data from a text file in HDFS storage.
- (ii) Fio-reader, Fio-writer and Fio are applications (programs in an I/O benchmark tool [5]) for

synchronous reading, writing data and process data in the disk drive shared with HDFS data blocks.

Also, applications TestDFSIO, Grep, FkmerG and TeraSort are used as well. TestDFSIO is a MapReduce-based, HDFS-stress testing benchmark. Grep finds matching strings from text files and counts the occurrence of the strings. Join is the program to combine data sets. FkmerG (based on WordCount) computes the frequency of k -mers of bacterial genomes³. TeraSort is a well-known Hadoop benchmark for testing both I/O-stressed and compute-intensive problems [8].

The following I/O performance metrics were used in the experiments:

- *Avg_read* (MB/s): The average throughput of reading HDFS data, that is calculated by the ratio of the total amount of HDFS read data over the total amount of time used by the threads for reading data from the disks.
- *Avg_write* (MB/s): This metric has the same definition as *Avg_read*, but for write activities.
- *blkio_delay* (ms): The average delay that an application spends waiting for a disk block. This measure is calculated by the ratio of the sum of delays experienced by an application over the total block I/O counts.
- *Runtime* (s): The overall time to run an application.

For each scenario, the average value of ten repeated measurements is calculated. The buffer cache was dropped before each measurement and was removed at the end.

4.2 Controlling I/O activities in Mesos

The disk I/O contention may occur when multiple tasks concurrently read or write data from the same disk. Figure 7 shows captured read data rate of Spark TeraSort and FkmerG on Mesos when running successively or in parallel. It is worth mentioning that FkmerG executors can run up to two tasks, while TeraSort executors can run up to three tasks in parallel. The HDFS block size is 256MB. There is a significant degradation in the disk I/O performance of the

³The data are prepared from NCBI Bacteria Genome Database [7] with the GkmerG tool [14]

Fig. 7 HDFS read data rates of FkmerG and TeraSort, Mesos environment

FkmerG if TeraSort is used, as shown in Fig. 7b. Figure 8 depicts the HDFS read rates when FkmerG and the TeraSort are executed in parallel. Figure 8a presents the data rate when a rate limit of 10 MB/s per task is set for both FkmerG and TeraSort. Similarly, Fig. 8b plots the read data rate when FkmerG was prioritized. Clearly, the I/O data rate performance of the FkmerG application can be provisioned with our solution.

Figures 9 and 10 plot the measured I/O performance metrics of the Spark FkmerG and TeraSort applications with/without an I/O greedy application. Note that the FkmerG and TeraSort executors were configured to run up to four tasks in parallel and the HDFS data block size is 256MB. When a specific workload (e.g., FkmerG or TeraSort) needs to be executed in a specific timeframe, administrators can choose our solution to limit the I/O contention of I/O

greedy applications. In this experiment, the maximum throughput of I/O greedy applications is limited to 20 MB/s. For example, FkmerG achieves a read rate of 7.48 MB/s when an HDFS-writer is run with the throttled I/O rate in parallel (as for comparison, FkmerG provides the read rate at 1.75 MB/s, when an HDFS-writer is run in an uncontrolled environment, until FkmerG achieves the read rate at 8.3 MB/s when there is no contention).

The experimental results show that the control of I/O-greedy applications is beneficial, in terms of the average delay for an I/O disk block for typical applications. As *CGroups* does not support the control of writing asynchronous data to disks, the I/O throughput of Fio-writer cannot be limited. Therefore, it is recommended to locate the HDFS storage in separate disks when the Hadoop cluster is shared with other applications [24].

Fig. 8 HDFS read data rates of FkmerG and TeraSort, Mesos environment

Fig. 9 Performance of the Spark FkmerG application with/without an I/O-greedy application in Mesos

Fig. 10 Performance of the Spark Terasort application with/without an I/O-greedy application in Mesos

Fig. 11 Performance of the MapReduce FkmerG application with/without an I/O-greedy application

Fig. 12 Performance of the MapReduce Terasort application with/without an I/O-greedy application

Fig. 13 Captured HDFS read data rate of the containers of MapReduce TestDFSIO and MapReduce Grep in a same worker machine, YARN environment

4.3 Controlling I/O Activities in YARN

The tool limits the data bandwidth of the Fio-reader and Fio-writer with the block size of 256MB to the maximum rate of 20 MB/s, while the LTC TCP/IP-based controller is used to control the rate of HDFS-reader and HDFS-writer to the maximum value of 20 MB/s. The applications are configured to execute up to four map tasks in parallel.

Figures 11 and 12 present the measured results of MapReduce FkmerG and TeraSort, using our approach to control disk I/O. The results demonstrate the impact of our approach for MapReduce workloads, except for the experiments with the Fio-writer. For example, the average throughput of TeraSort increased from 5.69 MB/s or 5.58 MB/s to 25.25 MB/s or 24.94 MB/s when applying I/O control to the HDFS- reader or HDFS-writer, respectively. Similar observations can be obtained for other MapReduce applications (Grep, TestDFSIO and Join).

The I/O contention is shown in Fig. 13a, which plots the captured throughput of the containers of MapReduce TestDFSIO and MapReduce Grep applications that were launched at the same time.

In this experiment, TestDFSIO (to process two files with the size of 2GB) and Grep (to process 1GB data with the block size of 256MB) were configured to execute up to two map tasks in parallel. Thus, four

Table 1 Execution times of two applications when running alone

Applications	Execution time (s)
Streaming Application	
Data ingestion (Kafka+Flume)	178
Processing data (MapReduce Join)	84
Batch Application	
Data ingestion (HDFS-writer)	236
Processing data (MapReduce Grep)	209

Fig. 14 Data processing phases in a typical use case

containers can try to access the same DataNode at any moment. In this case, FkmerG and TeraSort executors were configured to run up to two or three tasks in parallel, as described in Section 4.2 to shape the HDFS traffic of these applications. Figure 13b plots the captured results when the limited rate per task of TestDFSIO and Grep was set to 10 MB/s and 40 MB/s, respectively, while Fig. 13c plots the captured data when TestDFSIO was preferred with controlled data rate at 40 MB/s and Grep's rate is set to 10 MB/s. This case demonstrates that the HDFS read data rate of MapReduce applications can be appropriately shaped using our approach. Furthermore, the throughput of a container seems to be more predictable compared to the results in case of a limiting rate.

4.4 A Typical use Case

In this section, we present a typical use case at mobile network operators where our tool can provision the data rate of deadline-bounded applications. In

a computing cloud at mobile network operators data processing jobs consist of the data ingestion phase (prepare the acceptance of data from sources and stored in HDFS) and the processing phases as illustrated in Fig. 14. These applications are executed with periodicities and the execution should be finished by deadlines. Two applications are assumed.

- In a streaming application, events are continuously streamed from an external source with Kafka⁴ message bus and a Flume⁵ agent into HDFS. A MapReduce Join job is started at the beginning of each period to process the streaming data of the previous period. Thus, both data ingestion and Join job must be finished within 3 minutes. Note that the ingestion data of the streaming application (around 5 MB/s) is very small compared to the full copy rate of the HDFS-writer (around 100 MB/s)
- In a batch application, the data must be copied into HDFS, using an HDFS-writer tool from an external machine, which is not a part of the Hadoop cluster. A MapReduce Grep job is started after finishing the data ingestion phase.

The run times of two applications without competitions are presented in (Table 1). The streaming application is to be executed with the periodicity of 3 minutes, while the data processing job submitted by

Table 2 Execution time of the streaming application when running in parallel with the batch application

Phase	Execution time (s)
Run in parallel with HDFS-writer	
Data ingestion (Kafka+Flume)	204
Processing data (MapReduce Join)	325
Run in parallel with Grep	
Data ingestion (Kafka+Flume)	182
Processing data (MapReduce Join)	99

⁴A distributed streaming platform <https://kafka.apache.org/> can be used by applications to stream data

⁵A tool <https://flume.apache.org/FlumeUserGuide.html> can be used to collect, aggregate data.

Fig. 15 Processing pipelines for two applications with the HDFS rate enforcement tool

the batch application should be carried out for the periodicity of 12 minutes.

For each period, the streaming application must process around 900 MB data of events. Thus there are 2 Map tasks in each MapReduce Join job. The batch application must process 20 GB data, generated by HDFS-writer with the block size of 512 MB. Thus, there are 40 Map tasks in each MapReduce Grep job. The Capacity scheduler of YARN is configured so that the batch application can run up to 5 Map tasks in parallel, while the streaming application can run up to 2 Map tasks in parallel. When both MapReduce Grep and Join run in parallel, then the Join job can have a higher priority to run its Map tasks.

As both MapReduce Join and HDFS-writer are write-I/O intensive, they have a significant impact on the process of streaming data. It can be observed in Table 2 that the requirements of the streaming application cannot be guaranteed when the HDFS-writer copies the data to the HDFS with the full rate.

The proposed framework was used to enforce HDFS read/write rates of HDFS-writer and MapReduce jobs. The writing rate of HDFS-writer is limited to 60 MB/s. The HDFS write/read rate of each Grep and Join map tasks are limited to 20MB/s and 40 MB/s, respectively. The HDFS write/read rate of reduce tasks is 50 MB/s for both jobs. Therefore, we can organise the processing pipelines for two applications as in Fig. 15 to guarantee the deadlines.

5 Conclusion

In this study, we have demonstrated that free competition profoundly degrades the performance of applications in computing frameworks. We have proposed software components (available at <https://github.com/dohona/hadoop>), a set of functionalities to enable

monitoring of the I/O capability (disk and network) of the physical infrastructure and the control of I/O competition. Our experimental results have shown that our controlling tool that can easily be integrated into YARN and Mesos environments as well can minimize contention situations and I/O data rate overload of applications in shared clusters. It is worth emphasizing that our tool can be used to provision the disk I/O and the network bandwidth for containers in a cloud as well.

Acknowledgments We would like to thank Noémi Wagner for the language check and for the constructive review of the manuscript. We greatly appreciate Editors and anonymous reviewers whose constructive comments were very useful for the improvement of the presentation.

The research reported in this paper has partially been supported by the National Research, Development and Innovation Fund (TUDFO/51757/2019-ITM, Thematic Excellence Program). The research was partially supported by the Higher Education Excellence Program of the Ministry of Human Capacities in the frame of Artificial Intelligence research area of Budapest University of Technology (BME FIKP-MI/SC). The research of N. H. Do is supported by the OTKA K123914 project.

Funding Information Open access funding provided by Budapest University of Technology and Economics (BME).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Apache Hadoop. <http://hadoop.apache.org>. Accessed: 2017-06-05
2. Apache Mesos. <http://mesos.apache.org/>. Accessed: 2017-12-17
3. Apache Spark. <http://spark.apache.org/>. Accessed: 2017-12-17
4. CGroups Blkio Controller. <https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt>. Accessed: 2019-12-18
5. Fio - an I/O tool for benchmark and stress/hardware verification. <https://linux.die.net/man/1/fio>. Accessed: 2015-03-4
6. Linux Advanced Routing and Traffic Control: HOWTO. <http://lartc.org>. Accessed: 2017-04-10
7. NCBI Bacteria Genome Database. ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Bacteria/. Accessed: 2017-11-05
8. TeraSort benchmark for Spark. <https://github.com/ehiggs/spark-terasort>. Accessed: 2017-04-05
9. Amamou, A., Bourguiba, M., Haddadou, K., Pujolle, G.: A Dynamic Bandwidth Allocator for Virtual Machines in a Cloud Environment. In: 2012 IEEE Consumer Communications and Networking Conference (CCNC), pp. 99–104 (2012)
10. Do, N.H., Do, T., Tran, T.X., Farkas, L., Rotter, C.: Data I/O Provision for Spark Applications in a Mesos Cluster. In: 2016 19Th International Conference On Intelligence in Next Generation Networks (ICIN) (2016)
11. Do, T.V., Vu, B.T., Do, N.H., Farkas, L., Rotter, C., Tarjanyi, T.: Building Block Components to Control a Data Rate in the Apache Hadoop Compute Platform. In: 2015 18Th International Conference On Intelligence in Next Generation Networks (ICIN), pp. 23–29 (2015)
12. Drăgan, I., Iuhasz, G., Petcu, D.: A scalable platform for monitoring data intensive applications. *J. Grid Comput.* **17**(3), 503–528 (2019)
13. Enes, J., Cacheiro, J.L., Expósito, R.R., Touriño, J.: Big data-oriented PaaS architecture with disk-as-a-resource capability and container-based virtualization. *J. Grid Comput.* **16**(4), 587–605 (2018)
14. Karimi, R., Hajdu, A.: HTSFinder: Powerful Pipeline of DNA Signature Discovery by Parallel and Distributed Computing. submitted to the *Journal of Evolutionary Bioinformatics* (2015). <http://www.inf.unideb.hu/~hajdua/HTSFinder.html>
15. Kc, K., Freeh, V.: Dynamically Controlling Node-Level Parallelism in Hadoop. In: 2015 IEEE 8Th International Conference on Cloud Computing (CLOUD), pp. 309–316 (2015)
16. Ko, B.-M., Lee, J., Jo, H.: Toward enhancing block I/O performance for virtualized hadoop cluster. In: Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14, pp. 481–482. IEEE Computer Society (2014)
17. Kumar, K.A., Konishetty, V.K., Voruganti, K., Rao, G.V.P.: CASH: context aware scheduler for hadoop. In: Proceedings of the International Conference on Advances in Computing, Communications and Informatics, ICACCI '12, pp. 52–61. ACM (2012)
18. Liao, J., Yin, D., Peng, X.: Block I/O scheduling on storage servers of distributed file systems. *J. Grid Comput.* **16**(2), 299–316 (2018)
19. Malensek, M., Pallickara, S.L., Pallickara, S.: Alleviation of Disk I/O Contention in Virtualized Settings for Data-Intensive Computing. In: 2015 IEEE/ACM 2Nd International Symposium on Big Data Computing (BDC), pp. 1–10 (2015)
20. Murthy, A.C., Vavilapalli, V.K., Eadline, D., Niemiec, J., Markham, J.: Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2, 1st edn. Addison-Wesley Professional (2014)
21. Park, J.K.: Improving the Performance of HDFS by Reducing I/O Using Adaptable I/O System. In: 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), pp. 3139–3144. <https://doi.org/10.1109/ICEEOT.2016.7755280> (2016)
22. Recommendation ITU-T G.1000: Communications Quality of Service: A Framework and Definitions. International Telecommunication Union (2001)
23. Spivak, A., Nasonov, D.: Data Preloading and Data Placement for MapReduce Performance Improving. *Procedia Comput. Sci.* **101**, 379–387 (2016). 5Th International Young Scientist Conference on Computational Science, YSC 2016, Krakow
24. Tran, X.T., Do, T.V., Do, N.H., Farkas, L., Rotter, C.: Provision of Disk I/O Guarantee for MapReduce Applications. In: 2015 IEEE Trustcom/BigdataSE/ISPA, Vol. 2, pp. 161–166. IEEE, Helsinki (2015)
25. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: apache hadoop YARN: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13, pp. 5:1–5:16. ACM, New York (2013)
26. White, T.: Hadoop: The Definitive Guide, 4th edn. O'Reilly Media Inc (2015)
27. Xu, Y., Zhao, M.: IBIS: Interposed Big-data I/O Scheduler. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC '16, pp. 111–122. ACM, New York (2016)
28. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache Spark: A unified engine for big data processing. *Commun. ACM* **59**(11), 56–65 (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.