POLITECNICO DI MILANO

Facoltà di Ingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in

Computer Science and Engineering

xSpark: Managing Concurrent QoS-constrained Big Data Applications through Dynamic Resource Provisioning

Supervisor:

PROF. SAM JESUS GUINEA MONTALVO

Assistant Supervisor:

DR. GIOVANNI QUATTROCCHI

Master Graduation Thesis by:

SIMONE RIPAMONTI

Student Id n. 849786

Academic Year 2016-2017

COLOPHON

This thesis has been written in LaTeX using TeXstudio and is based on the template "*Classic Thesis at DEIB*" available here:

https://github.com/Lordmzn/ClassicThesis-at-DEIB

Ai miei genitori

— Simone

RINGRAZIAMENTI

Con questo lavoro si conclude la mia avventura nel mondo delle lauree magistrali al Poli.

Voglio prima di tutto ringraziare chi ha reso possibile la stesura di questa tesi, cioè il relatore Prof. Sam Guinea e il correlatore Dr. Giovanni Quattrocchi, entrambi si sono dimostrati disponibili e carichi di consigli, facendomi appassionare al loro lavoro.

La mia famiglia ha giocato un ruolo fondamentale, mi ha permesso di svolgere il mio percorso di studi senza preoccupazioni, fornendomi tutto il supporto necessario per trascorrere al meglio questa fase della mia vita. Un abbraccio forte ai miei genitori, Marco e Stefania, e un in bocca al lupo per il suo percorso a Roberto, un amico oltre che un fratello. Un bacione a tutti i miei nonni, anche a quelli che oggi non potranno essere qua.

Grazie a Francesca ♡, che mi è stata accanto e mi ha dato la forza di affrontare quest'ultimo ostacolo, spronandomi a dare il massimo, anche quando mi sembrava troppo in salita. Fondamentali sono stati i consigli lessico-grammaticali, che mi hanno permesso di scrivere al meglio le varie sezioni di questa tesi e non solo. Spero di poterti essere di supporto tanto quanto tu lo sia stata per me.

Grazie ad Alberto, Fulvio e Luca, per aver affrontato con me questi anni di università e avermi dato la possibilità di stemperare la tensione prima degli esami. Spero in un lunedì del sushi anche negli anni a venire.

Grazie a tutti i miei amici, vecchi e nuovi, per avermi sostenuto in ogni momento del percorso di studi e avermi dato la possibilità di trascorrere bellissimi momenti insieme. Un ringraziamento particolare a Umberto, amico fidato.

Simone

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

OS operating system

JVM Java Virtual Machine

EDF Earliest Deadline First

FIFO First In First Out

PI Proportional plus Integral

VM virtual machine

HDFS Hadoop Distributed File System

RDD Resilient Distributed Dataset

JT	Job Tracker
TT	Task Tracker
RM	Resource Manager
NM	Node Manager
AM	Application Master
NN	Name Node
DN	Data Node
CLC	container launch context
DAG	Directed Acyclic Graph
I/O	input/output
LXC	Linux Containers
MPI	Message Passing Interface
API	application programming interface
SQL	Structured Query Language
vCPU	virtual CPU
SSD	Solid State Disk
TPC	Transaction Processing Performance Council
QoS	Quality of Service

ABSTRACT

Cloud computing is based on computing power and storage virtualization, obtained using an infrastructure composed by abstract hardware and software, accessible on the Internet. The cloud enables big data processing for enterprises of all sizes. Big data are a massive amount of structured and unstructured data, they are so large that they can be difficulty processed using traditional database and software approaches. When working with large datasets, it becomes difficult to create, manipulate and manage these data, in particular it becomes a problem to search and analyze the data. Big data applications provide new challenges in searching and enforcing relevant quality of services, in particular users may be interested in quantifying and constraining the execution time of each of the applications' runs. One of the most commonly used cluster computing framework for big data analytics is Apache Spark, which provides a fast and general data processing platform, that allows quick in memory computation still being fault-tolerant. Spark computation is based on RDDs, a data abstraction, and DAGs, that represents the data manipulation process. xSpark, developed at Politecnico di Milano, propose an extension of Spark framework that offers fine-grained dynamic resource allocation using lightweight containers. xSpark allows users to force the duration of the execution of an application by specifying a deadline, this is possible thanks to the runtime allocation of resources requested during the execution by xSpark's control loop, that is composed by a centralized heuristic part and a distributed local control theoretical one. This thesis has the goal of extending the work done with xSpark, in particular by supporting the execution of multiple applications in parallel, trying to satisfy the quality of service requested by the users. This has been successfully done by extending the controller model and by taking into account the existence of multiple application running in parallel that may contend the available resources. In absence of resource contention, applications behave as if they were running alone in the cluster, meanwhile in presence of contention we are able to reduce the number of deadline violations picking the right strategy.

SOMMARIO

Il cloud computing è basato sulla virtualizzazione della potenza di calcolo e dell'archiviazione dei dati, ottenuto tramite un'infrastruttura di dispositivi hardware e software astratti e accessibili tramite Internet. Il cloud permette ad aziende di qualsiasi dimensione di eseguire applicazioni big data. I big data vengono descritti come un'enorme mole di dati strutturati e non, che sono talmente grandi da essere difficilmente elaborabili tramite basi di dati e approcci software tradizionali. Quando si lavora con grandi set di dati, diventa difficile creare, manipolare e organizzare questi dati, in particolare ricercare e analizzare questi dati diventa un problema. Uno dei principali framework di cluster computing per analisi di big data è Apache Spark, che offre una piattaforma per elaborare dati in modo veloce e generalizzato, permettendo una rapida computazione mantendo i dati in memoria e rimanendo allo stesso tempo tollerante agli errori. L'esecuzione di Spark è basata su RDD, un'astrazione dei dati, e DAG, che rappresenta il processo di manipolazione dei dati. Le applicazioni big data pongono nuove sfide nel cercare e imporre qualità del servizio rilevanti, in particolare gli utenti sono interessati nel quantificare e limitare la durata delle singole esecuzioni di queste applicazioni. xSpark, sviluppato presso il Politecnico di Milano, propone un'estensione del framework Spark in grado di offrire allocazione dinamica delle risorse a grana fine usando i container. xSpark permette agli utenti di forzare la durata dell'esecuzione delle applicazioni specificando una deadline, questo è reso possibile tramite l'allocazione delle risorse richieste durante l'esecuzione dal controllore presente in xSpark, composto da una parte euristica centralizzata e da una parte distribuita fondata sulla teoria del controllo. Questa tesi ha l'obiettivo di estendere il lavoro fatto con xSpark, in particolar modo si vuole supportare l'esecuzione di più applicazioni in parallelo, cercando di rispettare le qualità del servizio richieste dagli utenti. Questo è stato reso possibile estendendo il modello del controllore e tenendo in considerazione la possibile esistenza di più applicazioni che vengono eseguite in parallelo, e che quindi potrebbero contendersi le risorse disponibili. In assenza di contesa di risorse tra le applicazioni, esse si comportano come se stessero eseguendo da sole nel cluster, mentre in presenza di contesa, siamo in grado di ridurre il numero di violazioni della deadline scegliendo la strategia più adeguata.

INTRODUCTION

Cloud computing has evolved to the point that it has become a popular and universal paradigm of service oriented computing, where infrastructures and solutions are offered as a service. Cloud has revolutionized the way a computing infrastructure is abstracted and used. Some of the feature that make cloud computing desirable are *elasticity* (e.g., on demand scaling), *pay-per-use*, *no upfront investment* (or a very little one), *low time to market* and *transfer of risk*.

Big data is a word that is used to describe large amounts of data, that might be structured, semi-structured or unstructured. We call big data those data that cannot be handled using traditional databases or software technologies. The term big data is originated from the web companies that needed to handle loosely structured or unstructured data. Today, every second 7.500 Tweets are sent, 800 Instagram photos are uploaded, 1.300 Tumblr posts are created, 2.800 Skype calls are done, 60.000 Google searches are performed and 2.500.000 emails are sent [5]. All this data is collected and analyzed. There are many properties associated with big data: *volume*, *variety*, *velocity*, *variability* and *value*. Storing and processing big volumes of data requires scalability, fault tolerance and availability [29]. Through hardware virtualization, cloud computing provides all the requested characteristics. Cloud computing enables big data to be available, scalable and fault tolerant. Big data is also a business opportunity, many companies started to focus on delivering big data applications as a service, such as Cloudera[1], Teradata[2] and many others. Big data applications have the goal of transforming, aggregating and analyzing large amounts of data in a easy and efficient way. Specialized frameworks have the goal of transforming these applications in atomic parts, that can be executed in a distributed cluster of physical or virtual machines. The level of parallelism that we can achieve is limited by the number of machines and also by the synchronizations requested between the data, such as aggregations and grouping. The first example of this paradigm was the map/reduce programming model, now more advanced solutions, such as Apache Spark [2] and Apache Tez [3], provide a greater flexibility and allow building more complex applications using a DAG based structure.

One of the most commonly used cluster computing framework for big data analytics is Apache Spark [30]. It was originally developed by researchers at UC Berkley AMPLab. Apache Spark is a distributed compute framework for easy, at-scale, computation [10].

1 Cloudera: www.cloudera.com
2 Terradata: www.teradata.com

Spark provides a fast and general data processing platform, letting users execute programs 100x faster in memory or 10x faster on disk than Hadoop, indeed in 2014 it won the Daytona GraySort contest as the fastest open source engine for sorting a petabyte [42]. Spark is fault-tolerant and is designed to run on commodity hardware, it generalizes the two stage MapReduce to support arbitrary DAG. The main advantage of Spark with respect to previous cluster computing frameworks is the fast data sharing between operations, for example Apache Hadoop requires intermediate data to be written on disk in order to be accessible by the following operations, Spark instead allows to execute in-memory computing. Spark offers a quick way of writing code by means of high-level operators provided in the API: Spark Core, Spark SQL, Spark Streaming, MLlib (machine learning), GraphX (graph). Spark integrates well with various storage systems, including Amazon S3, Hadoop HDFS and any POSIX-compliant file system. Spark provides its own cluster manager, but it can also run on clusters managed by Hadoop Yarn or Apache YARN. Spark is often used for in-memory computation, but it is also capable of handling workloads whose size exceeds the aggregate cluster memory.

Quality of Service (QoS) notion in big data application change according to the type of application. Interactive applications are usually assessed in terms of *response time* or *throughput*, their fulfillment depends on the intensity and variety of the incoming requests. Big data applications might require a single batch computation on a very large dataset, thus QoS must consider only the execution of a single run. In this domain QoS is often states as *deadline*, that is the desired duration of the computation. Many factors influence the duration of an application execution. A resource allocation problem is due to the fact that different applications might have a different structure, applications can run in contexts that have different available resources and also the input datasets might have different sizes. A scheduling problem instead is due to the fact that other applications might be running on the same hardware, thus not all the resources are available for an application execution.

Satisfying deadline based QoS constraints is a problem related to resource allocation, since the amount of allocated resources determines the duration of the execution of a Spark's application. The simplest option available on all cluster managers is static partitioning of the resources, in this way each application is given a maximum amount of resources it can use, and holds them for the whole execution. Memory sharing across applications is currently not provided. Spark also provides a mechanism to dynamically adjust the resources an application occupies according to the workload, applications may give resources back to the cluster if no longer used and may reacquire them again when there is demand, this is particularly useful if multiple application share resources in the Spark cluster. This is disabled by

default, but it is available on all cluster managers. A problem of this approach is the granularity, since it is performed on the granularity of the executor.

In this thesis we investigate the problems related to the scheduling of concurrent DAG-based big data applications, taking into account QoS constraints. In this field, the state-of-the-practice big data framework is Apache Spark. This thesis is based on an extension of Spark, called xSpark, which offers dynamic resource allocation and enforces QoS constraints.

The previous work on xSpark addressed the resource allocation problem, in order to meet user defined deadlines in Spark. xSpark is a Spark extension that offers optimized and elastic provisioning of resources. This is obtained by using nested control loops. A centralized loop is implemented on the master node, it controls the execution of different stages of an application. Multiple local loops, one per executor, focus on task execution. xSpark exploit an initial profiling application execution in order to create an enriched DAG of the application, storing informations about the stages. At runtime, the annotated DAG is used to understand how much work has already been done and how much work still needs to be done. Since we need all executions of the same application to use the same DAG, xSpark requires applications to not contain branches or loops, which might be resolved in different ways at runtime. The centralized control loop is activated before the execution of each stages and uses a heuristic to assign a deadline to the stage and calculate the required CPU cores needed to satisfy it, using the provided enriched DAG and the user requested deadline. Many factors can influence the actual performances and invalidate the prediction, local control loops have the objective of counteracting against those imprecisions, by dynamically modifying the amount of CPU cores assigned to the executors during the execution of a stage. A control theory algorithm determines the amount of CPU cores to be allocated to the executor for the next control period. Docker is used in order to tune the number of CPU cores allocate to the executors, which are run inside lightweight containers. xSpark is able to use less resources than native Spark and can complete executions with a less than 1% error in terms of set deadlines.

The problem of meeting a particular deadline depends on a resource allocation problem (addressed in xSpark work) and a scheduling problem, since other applications may be in execution on the same hardware.

The scheduling problem on xSpark prevents executing multiple applications at the same time. This is due to the fact that there is no policy to share resources across applications. This problem is a relevant one, in particular when executing applications with a long deadline. Having a long deadline, intended as greater than the application execution time when executed in native Spark, will lead to have a low

average resource utilization. This has the effect of wasting resources, since we are not able to exploit them by running another application in parallel. Our objective is to give concurrent applications the capability of requesting resources in an elastic way, which means that every application executes as if it is the only running application at the time and thus it will try to use all the system resources regardless the presence of other applications. As one can expect, this behavior can lead to resource contention between different application, a way to solve contention needs to be proposed. In the end, it is important to check that it is possible to solve both the resource allocation and the scheduling problem at the same time. A relevant limitation in Spark is the absence of a way to resize the memory that has been assigned to an executor. Resizing the JVM heap of an executor is impossible by design, it is required to restart the process. Instead, off-heap allocated memory could be resized since it does not reside on the Java process, but this feature is absent in native Spark. Being able to resize the memory assigned to an executor would allow to have an higher memory utilization, reducing resource waste.

This thesis adapts xSpark to the concurrent applications context. The two main modifications of the previous work are related to the application scheduling and the dynamic use of off-heap memory. The distributed control loop of xSpark has been modified in order to handle the presence of multiple applications' executors running on the same worker node. In order to do so, we needed to investigate different ways to solve the resource contention state, since we are interesting in allocating no more than the available resources, in order to have a predictable behavior of the application execution. Different strategies for solving resource contention have been proposed, which may take into account the priority of an application according to its deadline (based on Earliest Deadline First approach) or might be based on a weight calculated using some characteristics of the application, such as the requested cores or its speed. A tuning parameter is available in order to be more tolerant of the presence of other applications running in the system, in particular its purpose is forcing the allocation of the entire cluster CPU cores in order to anticipate the completion of stage execution with respect to their deadlines. This has the advantage of better tolerating the future presence of new applications running in the system, which might slow down the application execution. It is important to remember that this might cause the anticipation of the application completion, with respect to the defined deadline, if no contention state arises, which as a consequence might cause a large error between the expected completion time and the resulting one. In order to be able to use the entire cluster resources, we introduced a way to dynamically resize the allocable off-heap memory assigned to an application. Off-heap memory size is updated every time the context changes, in particular when a new application is launched

or one terminates. This allows a single application to be able to allocate the entire resources of the cluster in terms of CPU cores and off-heap memory, and still be able to launch another application as if the previous one was not occupying all the resources.

This xSpark extension has been tested against a different set of benchmarking applications, that include machine learning, graph processing, simple aggregations and SQL queries. We used composite benchmarks to understand the behavior of the system, which are composed by a set of applications, each with a specified deadline and release time. From our experiments, we understood that the different resource contention solving strategies behave differently and are suitable in different contexts, for example when dealing with a strict deadline, we may choose a strategy that is based on Earliest Deadline First approach, when our goal instead is minimizing deadline errors instead, we may prefer to choose a weight based strategy. In order to be able to determine if a composite benchmark is feasible or not, in the sense that a deadline-aware implementation of Apache Spark would be able to satisfy the deadlines, we developed a tool that given the vanilla Spark execution logs of the applications in a composite benchmark, is able to determine a possible parallel execution behavior using an Earliest Deadline First approach. When a composite benchmark is feasible, this extension of xSpark is able to achieve errors under 5% using certain combinations of contention resolving strategies and tuning parameter. When a composite benchmark is infeasible instead, we are able to complete the same number of applications satisfying the deadline as the ones expected from the execution log based tool. We have also compared the execution of parallel applications in xSpark with serialized and parallelized vanilla Spark, proving that xSpark is able of outperforming serialized Spark with certain applications in terms of execution time and in general performing not worse than a statically partitioned Spark cluster. Latest experiments took into account the changes in performance related to the use of off-heap allocation, proving that off-heap allocation is a solution when disk swapping degrades execution performances, but in general has the effect of slightly increasing execution times due to an higher time requested to access stored data.

1 STATE OF THE ART

1.1 MAPREDUCE

MapReduce is a software framework introduced by Google in order to support the distributed computation of large dataset in cluster of computers. The framework is inspired by map and reduce functions used in functional programming, even though their purpose in the MapReduce framework is not the same of the original form. MapReduce library are available written in different programming languages. There are open source implementation of the MapReduce framework, for example Apache Hadoop.

The MapReduce framework is composed by different functions for each step:

1. Input Reader
2. Map Function
3. Partition Function
4. Compare Function
5. Reduce Function
6. Output Writer

The Input Reader reads the data from mass memory and splits the input in S different splits, with a fixed dimensions (e.g., 64 MB) that are successively distributed to M machines of the cluster that have the Map Function. The Input Reader has also the goal of generating a pair (key, value). The N machine of the cluster are divided in 1 master, whose goal is to detect idling slaves and assign them a task, and $N-1$ slaves that receive the tasks assigned by the master node. In total, M Map tasks and R Reduce tasks are assigned. A slave that has been assigned the $M-th$ task reads the content of the input, extracts the (key, value) pairs and send them to the Map function defined by the user, that generates zero or more (key, value) pairs as output. These pairs are buffered in memory. Periodically the buffered pairs are cached on disk and partitioned in R sections by the partition function. The addresses of the partitioned sections are sent to the master node which is responsible of rotating the location of the slaves that will process the Reduce function. Between the slave with the Map function and the one with the Reduce one, all the pairs are reordered in order to find the ones that point at the same value, and thus also

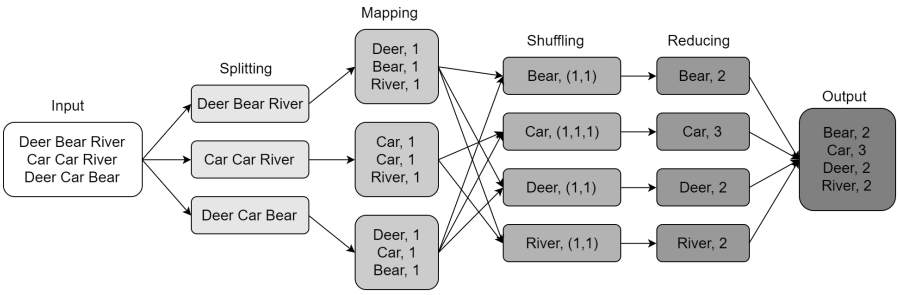


Figure 1.1: MapReduce word count job example. The goal of this job is to count the occurrences of the different words that are in the input text.

have the same key. The so called shuffling phase is the process that is used to transfer data from mappers to reducers. Once all the keys that point to the same value have been found using the compare function, a merge operation is performed. The sorting operation is useful because in this way the reducer can know when a new reduce task should start. For each of the keys, the associated slave iterates on all the keys, takes the values with the same key and then applies the Reduce function defined by the user, generating one or more element in output. The Output Writer has the goal of writing the results back on mass storage.

A sample word count application can be seen in Figure 1.1. The input is a document containing words, our goal is to compute the number of occurrence of each of the words in the document. Each Map task applies its function on a line of the document, emitting for each of the words in the line a pair ('word', 1). For example if the input line is "Dear Bear River", it is split into ["Dear", "Bear", "River"] and then mapped into [("Dear", 1), ("Bear", 1), ("River", 1)]. After shuffling the map results, the Reduce task receives a word and a list containing as many ones as the times the word appeared in the document, the reduce function will simply sum the ones in the list, emitting as a result the pair ('word', 'count'). For example, a reducer can receive the key "Bear" with list of values (1, 1), this is reduced into ("Bear", 2). Reducers results are then collected and stored in mass memory.

1.1.1 *Apache Hadoop*

Apache Hadoop is an open-source framework for distributed storage and processing of big datasets using MapReduce programming model.

Apache Hadoop MapReduce [13] cluster have a centralized structure composed by a single master Job Tracker (JT) and multiple worker nodes running Task Tracker (TT), as shown in Figure 1.2. JT main goal is organizing the job tasks on the slave nodes and continuously monitor the Task Trackers by means of heartbeats. Heartbeats provide a

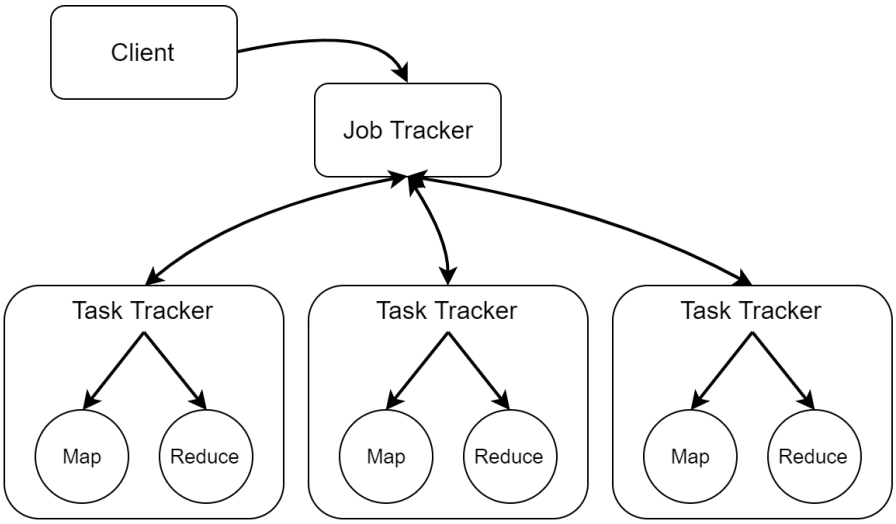


Figure 1.2: Hadoop MapReduce architecture.

way to retrieve informations about the liveliness of the slaves and to inspect the progress of the executions of the different tasks. In order to be fault tolerant, if a task execution fails, it is re-executed possibly on a different slave. JT has the role of the cluster manager, so it needs also to check the admissibility of the submitted MapReduce jobs. TT have the objective of running the assigned task. They reply to heartbeats in order to affirm their liveliness and to update the master about the progress of the assigned tasks. They are configured with a fixed number of map and reduce task slots.

As previously introduced, Apache Hadoop also offers a distributed file-system that stores data on different machine, providing an high aggregate bandwidth across the cluster. This functionality is called Hadoop Distributed File System (HDFS)[14]. It is highly fault tolerant and designed to be deployed on low cost hardware. HDFS exposes a filesystem namespace and allows user data to be stored in files and retrieved. Cluster structure is similar to the one of MapReduce cluster, with one master and multiple slaves, as we can see from Figure 1.3. The master is composed by a single Name Node (NN), that manages the file system namespace and regulates access to the files by clients. NN executes filesystem operations such as opening, closing, renaming files and directory, but the most important operation performed is keeping track of the mapping between blocks and Data Node (DN). Indeed a file stored in HDFS is split into one or more blocks, and those blocks are stored in the Data Node (DN). Data Node (DN) represent the slaves, they are usually one per node, and manage the storage that is attached to the node they are running on. They are responsible for serving read and write operation requests from the clients, but also can perform block creation, deletion and replication. Block replication is a significant way to improve fault tolerance.

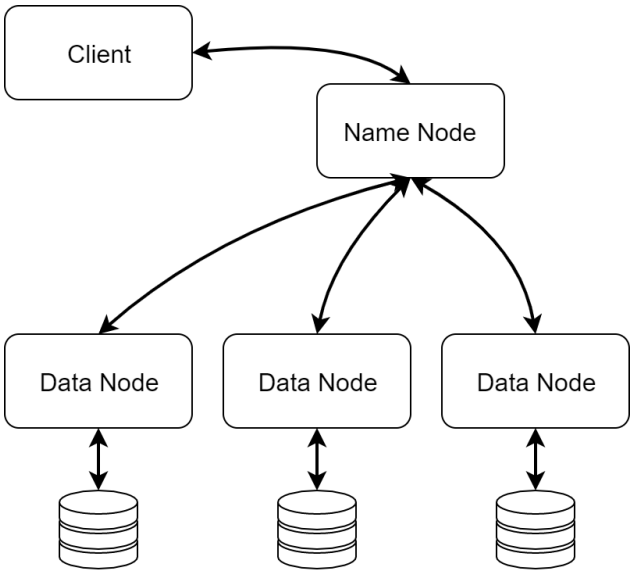


Figure 1.3: Hadoop HDFS architecture.

1.2 APACHE HADOOP YARN

Apache Hadoop YARN (Yet Another Resource Negotiator) is the next generation of Hadoop's compute platform [37]. The idea is to split the functionality of resource management and job scheduling and monitoring. This is achieved by having two different kind of daemons running, one global Resource Manager (RM) and a per-application Application Master (AM).

Resource Manager (RM) and Node Manager (NM) form the data computation framework (Figure 1.4). RM is the authority that manages resources among all the applications that are running in the system, meanwhile NM is the per-machine daemon who is responsible for managing containers, monitoring and reporting. The per-application AM has the goal of negotiating resources with RM and working with NM in order to execute and monitor tasks.

Resource Manager (RM) is composed by two components: *Scheduler* and *Applications Manager*.

The *Scheduler* is responsible for allocating resources to the various applications that are running, by taking into account constraints about capacity, queues, etc. It is a real scheduler in the sense that it does not perform monitoring or tracking of the application state. Moreover, it does not offer any guarantee that a failed that will be re-executed after an application or hardware failure. The *Scheduler* performs the allocation according to the resources that are requested by an application; this is based on the abstract notion of container which has elements as memory, CPU cores, disk and network bandwidth. There are pluggable policy that determine the repartition of resources among the different applications, for example we have the *Capacity*

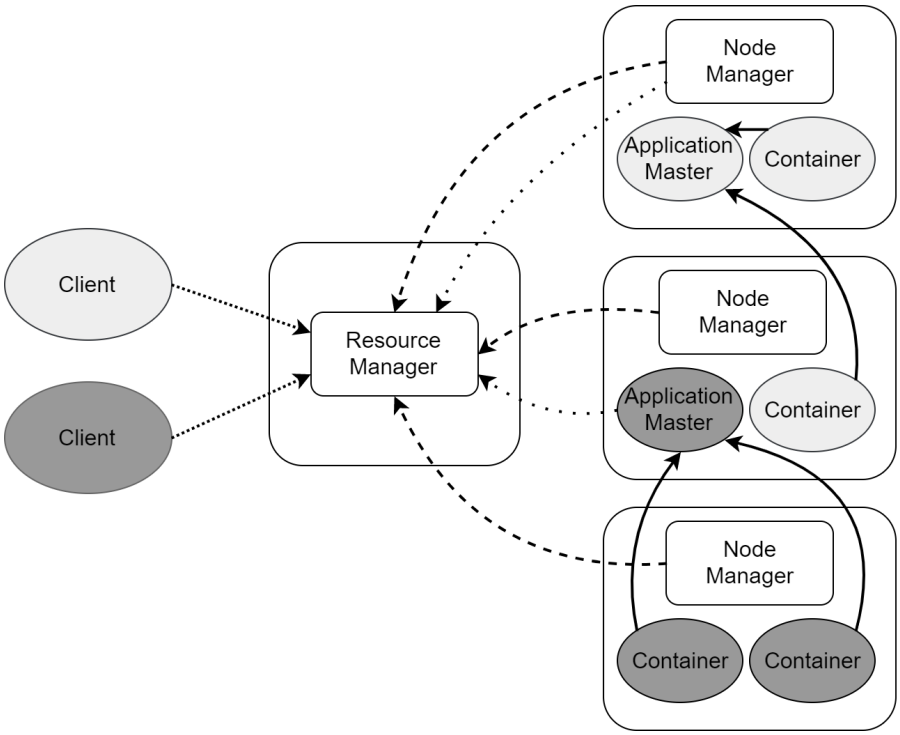


Figure 1.4: Hadoop YARN architecture.

Scheduler, designed for multi-tenant clusters, and the *Fair Scheduler*, that shares cluster resources fairly.

The *Applications Manager* is responsible of accepting the submission of a job, it negotiates the first container that will execute the AM and it offers a service that can be used to restart the AM in case of failure. The per-application AM has the goal of negotiating the needed containers from the Scheduler, track their status and monitor their progress.

The RM keeps a global model of the cluster state and thanks to the resource requirements reported by the running applications, it makes possible to enforce a global scheduling, but it is required to have an accurate understanding of the applications' resource requirements. In response to AM requests, the RM generates containers together with tokens that grant access to resources. An extension of the protocol allows the RM to ask back resources from applications, for example when cluster resources become scarce.

Application Master (AM) is the process that coordinates the execution of an application inside the cluster. It is important to remember that itself is run in the cluster, just like any other container. Periodically, an heartbeat is sent to the RM in order to confirm its liveliness and to update the Scheduler about its resource requests. After having modeled the application requirements, the AM codifies its preferences and constraints inside the heartbeat message. This informations are stored in the form of *Resource Request*, containing the desired num-

ber of containers (e.g., 100 container), the resources of each container (e.g., <2 CPU, 2 GB>), the locality preferences and the priority of this resource request with respect to the other ones of this application. When a container lease is received, the AM can choose to modify its execution plan in order to take into account the abundance or scarcity of resources.

Node Manager (NM) is the worker daemon in YARN, its purpose is to authenticate container lease, manage dependencies, monitoring the execution of containers and offer them a set of services. After having registered with the RM, the NM sends heartbeats in order to communicate its status and receives instructions from the RM. All the containers are described by a container launch context (CLC), that keeps track of all the environment variables, the dependencies, the security tokens, but also of the payloads needed by NM services and the commands that are needed to launch the process inside the container. After having validated the authenticity of the container lease, the NM configures the container with the specified resource constraints and initializes a monitoring subsystem. In order to launch the container, dependencies are copied into local storage. NM also has the duty of killings container upon a request from RM or AM, for example when a tenant is being evicted or when an application completes. Whenever a container exits, NM needs to clean the working directory. When an application ends, all the resources held by its container on all nodes are released. NM periodically checks the state of the physical machine and informs the RM of a possible unhealthy state.

1.3 APACHE MESOS

Apache Mesos is an open-source project used to manage computer clusters. The purpose of Mesos is to share cluster between different computing frameworks, such as Apache Hadoop or Message Passing Interface (MPI). The sharing increments the utilization of the cluster and prevents per-framework data replication. Mesos shares resources in a fine-grained way, allowing to achieve data locality. It presents a scheduling mechanism on two layer called *resource offers*. Mesos decides how many resources to offer to each of the running frameworks, meanwhile they decide how many resources to accept and which computation to execute on the granted resources.

New cluster computing frameworks continue to emerge, it is clear that finding a framework that is optimal for all type of application is almost impossible. We expect that organization would like to use different frameworks inside the same cluster, picking the best one according to the kind of application that they are going to execute. Two classic solution are: i) statically partitioning the cluster and executing one framework per partition; ii) allocate a set of VMs to each of the frameworks. Unluckily these solution do not achieve high utilization

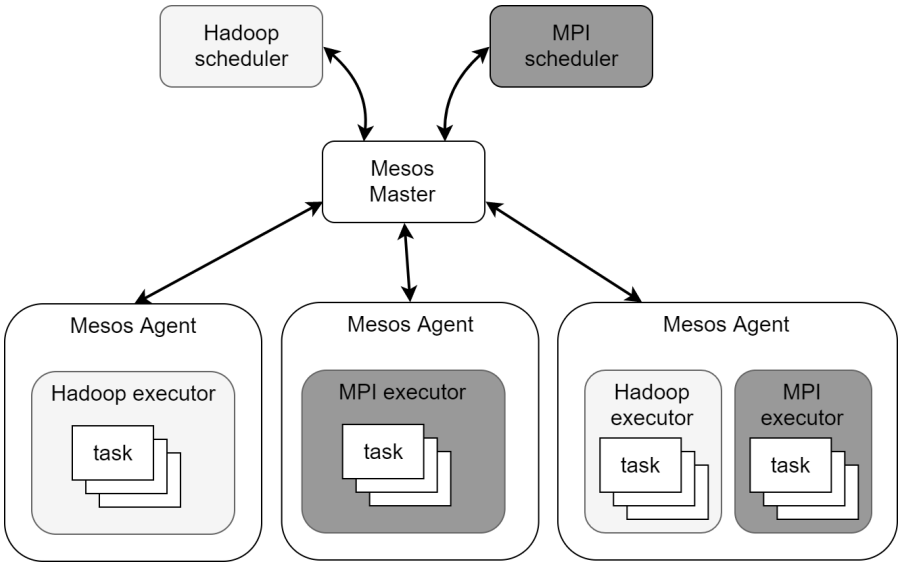


Figure 1.5: Apache Mesos architecture.

and efficient data sharing. The main problem is the different allocation granularity of these solutions and the one of the existing frameworks, for example Hadoop employs a fine grained resource sharing model, where nodes are divided into slots and each job is composed by short tasks that match the slots. The presence of short tasks allows us to achieve high utilization, as jobs can rapidly scale when new nodes are available. But it is not possible to achieve fine grained sharing across frameworks, because they have been developed in an independent way, and thus it is difficult to efficiently share the cluster among different frameworks.

Mesos delegates the control over the scheduling to the different frameworks. In this way it is possible to have the abstraction of the resource offers, that encapsulate a bundle of resources that the framework can allocate on a node in order to execute a task. Mesos decides how many resources to offer to each framework, this is based on policies, and the framework decides which resources to accept and which tasks to execute on them. Even though this approach does not lead to a globally optimum scheduling, it has been proved that it performs particularly well in practice, allowing the frameworks to obtain near perfect data locality. Mesos provides other benefits to its users, for example the possibility of runnin different instances of the same framework or even different versions.

Mesos is composed by a master process that manages slave daemons running on each cluster node and frameworks that run tasks on these slaves, as we can see from Figure 1.5. Master implements fine-grained sharing across frameworks using resource offers. Every resource offer is a list of free resources on the different slave nodes. The master decides how many resources to offer to each framework, according to some policy such as fairness or priority. Every frame-

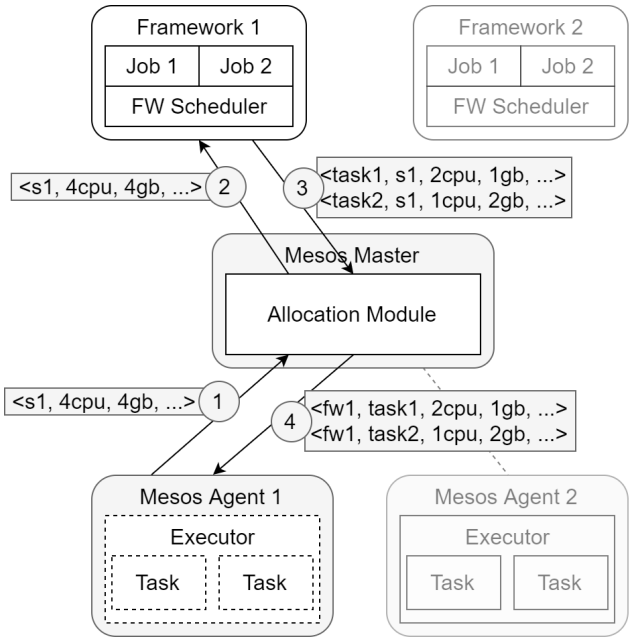


Figure 1.6: Apache Mesos Resource Offer example. 1) Mesos Agent 1 reports free resources to the Allocation Module; 2) Allocation Module offers resources to Framework 1 scheduler; 3) Framework 1 scheduler accepts resources and assign tasks; 4) Allocation Module launches tasks on the executor running in Mesos Agent 1.

work that is running on Mesos is composed by two components: a scheduler, that registers with the master in order to obtain the resource offers, and an executor process that is launched on the slave node in order to execute framework's tasks. While the master chooses how many resources to offer, the scheduler chooses which resources to use among those offered. When an offer is accepted, the scheduler sends to the master the description of the tasks that should be executed. The resource offer process is repeated every time tasks are finished and when there are new free resources. In order to maintain a light interface, Mesos does not ask the frameworks to specify their resource requirements or constraint, instead it gives them the possibility of refusing offered resources. Mesos allows frameworks to set up a set of filters, in the form of boolean predicates, specifying the conditions on which the framework will always refuse a proposal (e.g., providing a whitelist of nodes it can run on).

In Figure 1.6 we have an example of resource offer process.

1. Agent 1 reports to master that it has 4 CPUs and 4 GB of memory free. Master invokes its allocation module policy, which tells that framework 1 should be offered all the resources.

2. Master sends a resource offer describing the resources available on agent 1 to framework 1.

3. Framework's scheduler replies to master with informations about two tasks to run on agent 1, using the specified resources for the two tasks.

4. Master sends the tasks to agent 1, which allocates the appropriate resources to the framework's executor, that will launch the two tasks.

Resource allocation is performed by a pluggable allocation module, such that it is possible to meet different organization needs. The two basic allocation modules are fair sharing and strict priorities, similar to those available on Apache Hadoop. In the normal situation, Mesos does exploits the fact that the majority of the tasks are short and so it reallocates resources only when tasks end. This usually happens frequently and so a new launched framework can obtain its share quickly. The allocation module can also revoke tasks, killing them, but before doing so it concedes a grace period to the framework in order to terminate them properly. The allocation module chooses the policy to revoke tasks, it needs to take into account the fact that this might be of little impact on some framework (e.g., MapReduce), but it can be critical in frameworks that have interdependent tasks (e.g., MPI). For this reason, the allocation module exposes a guaranteed allocation for each of the frameworks, an amount of resources that the framework can allocate without the risk of losing tasks, this value can be retrieved by the framework using an API call. If the framework total allocation is under the guaranteed one, it has no risk of seeing its task killed, on the other hand instead, if the allocation is over the guaranteed one, any of its tasks can be terminated.

Performance isolation between frameworks executor running in the same slave is achieved by leveraging existing OS isolation mechanism. Since they are platform dependent, pluggable isolation modules are supported.

1.4 APACHE SPARK

Apache Spark is an open source framework for distributed computation [2], that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. With respect to the MapReduce paradigm, the in-memory multilevel primitives of Spark allow to have performances up to 100 times better in certain applications. Spark can work as standalone or on a cluster manager such as Apache Hadoop Yarn or Apache Mesos. It also needs a distributed storage, it can natively use HDFS and other solutions.

Spark has been designed as a unified engine for distributed data processing. Spark has a programming model that is similar to the one of MapReduce, but it is extended with the data sharing abstraction fo Resilient Distributed Dataset (RDD). Using this abstraction, a

wide range of processing workloads can be captured, including SQL, streaming, machine learning and graph processing.

The generality of Spark approach gives great benefits. First of all, all applications are easier to develop since there is a unified API. Secondly, it is a lot easier to combine processing task, with previous distributed computation framework we needed to write data to mass storage before using them in another engine, instead Spark can work multiple times on the same data, often keeping the in memory.

The programming abstraction at the foundation of Spark is Resilient Distributed Dataset (RDD), that are fault-tolerant collection of objects partitioned across the cluster that can be manipulated in parallel. The users create RDD by applying operations called "transformations", such as map, filter and group-by, on the data. RDD can be backed by a file obtained from an external storage. Spark evaluates the RDD in a lazy way, in order to allow finding an efficient plan to execute the computation requested by the user. In particular, every transformation operation returns a new RDD, that is the representation of the result of the computation, but the computation is not executed immediately. When an "action" is requested by the user, Spark check the entire graph of the transformation and uses it to create an efficient execution plan. For example, if there are many filter and maps in a row, Spark can fuse them together and execute a single operation. RDDs also offer an explicit support to perform data sharing among the computations, by default they are ephemeral but they can be persisted to disk or memory for rapid reuse. This data sharing is main difference between Spark and the previous computing models like MapReduce, because all the other operations that Spark can perform are similar to the ones of MapReduce. The data sharing allows to obtain huge speedups, up to 100 times, in particular when used to execute interactive query and iterative algorithms.

RDDs can also recover automatically from a failure. Traditionally, fault tolerance was in distributed computing was achieved by means of data replication and checkpointing. Spark instead uses a different approach called *lineage*. Each RDD keeps track of its transformation graph that has been used to generate the RDD and re-executes these operation on the base data in order to recover every lost partition. The data recovery based on lineage is significantly more efficient than replication in case of data-intensive workload. In general, recovering lost partitions is faster than re-executing the entire program.

Spark has been designed in order to support different external systems for persistent storage, usually it is used paired with a cluster file system like HDFS. Spark is designed as storage-system-agnostic engine, in order to make it easy to run computation against data from different sources.

Different high-level libraries have been developed in order to simplify the creation of programs that can run in Spark framework.

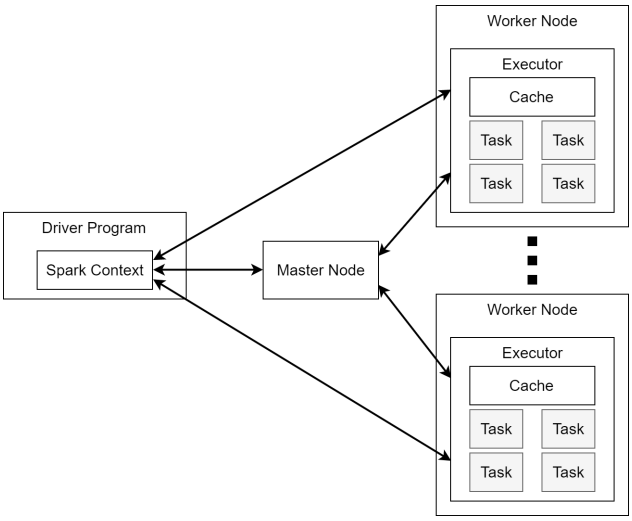


Figure 1.7: Spark Standalone architecture.

- SQL and DataFrames: support for relational queries, that are the most common data processing paradigm
- Spark Streaming: implements incremental stream processing using a model called "discretized streams", input data is split into micro batches
- GraphX: graph computation interface
- MLlib: machine learning library, more than 50 common algorithms for distributed model training

Spark architecture follows the master/worker paradigm (Figure 1.7). A master server accepts data and processing request, split them into smaller chunk of data and simpler actions that can be handled in parallel by the multiple workers. A Spark application is executed inside a driver program, that makes the user code executable on the computing cluster using a *SparkContext*. The driver program is responsible for managing the job flow and scheduling tasks that will run on the executors. The *SparkContext* will split the requested operations in tasks the can be scheduled for the distributed execution on the workers. When a a *SparkContext* is created, on each worker a new Executor process is created. An executor is a separate Java Virtual Machine (JVM) that runs for the entire lifetime of the Spark application, executes tasks using a thread pool and store data for its Spark application. Communication between the *SparkContext* and the other components is performed using a shared bus.

When an application is submitted to Spark, it is divided in multiple *jobs*. Jobs are limited by the presence of Spark actions within the application. Spark actions are those operations that return a value to the driver program after running a computation on the dataset. For each job, a Directed Acyclic Graph (DAG) is created in order to

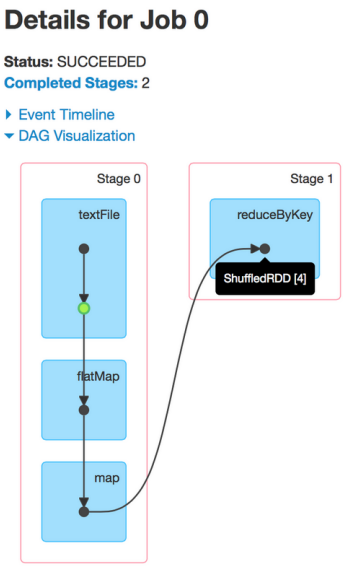


Figure 1.8: Spark word count application DAG example. The goal of this job is to count the occurrences of the different words that are in the input text.

keep tack of the RDDs that are materialized inside the job. DAG nodes represent the RDDs, meanwhile arcs represent transformations, that are those operations that create a new dataset from an existing one. The application steps inside a single job are further organized into *stages*, that are delimited by operations that require data reshuffling, that will inevitably break locality. Spark distinguishes between *narrow* transformations, that do not reshuffle data (e.g., `map`, `filter`), and *wide* transformations, that require data reshuffling (e.g., `reduceByKey`). Stages are also used to produce intermediate result that can be persisted in order to avoid re-computation. When all stages inside a job have been identified, Spark can determine which parallel task need to be executed for each stage, and then schedules them for operation on the executors. Spark creates one task for each of the partitions of the RDD that a stage receives in input.

Listing 1.1: Spark word count application example.

```
sparkContext.textFile("hdfs://...")
.flatMap(line => line.split(" "))
.map(word => (word, 1)).reduceByKey(_ + _)
.saveAsTextFile("hdfs://...")
```

In Figure 1.8, we can see a simple DAG representing the single job of the word count application presented in Listing 1.1 [26], the image has been obtained from Spark Web UI. Through a `textFile` operation, the input file is read from HDFS. Then a `flatMap` operation is applied to split each of the lines of the document into words. Following, another map is used to create ('word', 1) pairs. Finally, a *reduceByKey* op-

eration is performed in order to count the occurrences of each word. The blue boxes represent the Spark operations that the user calls in his code, meanwhile the dots represent the RDDs that are created as a result of these operations. Operations are grouped into stages, represented by the boxes with a red border. The job has been divided into two stages because the *reduceByKey* transformation requires the data to be shuffled. The green dot represents a cached RDD, in particular the data read from HDFS has been cached, in this way future computations on this RDD can be done faster since data will be read from memory instead of HDFS.

The default deployment of Spark is in standalone mode, using its embedded cluster manager without the need of running on to of another one. It is important to remember that the cluster manager is responsible for starting executor processes and determine where and when they will be run. Using Spark's embedded cluster manager might be a problem in terms of resource utilization when we want to execute different distributed applications together with Spark. Using a single cluster manager for different distributed applications has the advantage of providing a global view on which applications are running and which we want to execute inside the cluster.

Without a single cluster manager, we can have two main approaches in order to perform resource sharing and allocation:

- allowing every application to allocate all the resources in the cluster at the same time, this leads to an unfair situation of resource contention
- splitting the resource pool into smaller pools, one per application. In this way we will avoid resource contention but we will have a less efficient utilization of the resources, because some of the applications might request more resources than the ones in the pool, meanwhile some others are using less resources than the allocable ones in order to execute

A more dynamic way of allocating resources will led to a better resource utilization. Spark natively support executing on top of Apache Hadoop YARN and Apache Mesos cluster managers.

Spark supports the dynamic allocation of executors, also known as elastic scaling, this feature allows to add and remove Spark executors in a dynamic way in order to match the workload.

In traditional static allocation, a Spark application would allocate CPU and memory upon starting the execution, disregarding how much resources will effectively use later on. With dynamic allocation instead it is possible to allocate as much resources as they are necessary, in order to avoid wasting them. The number of running executors is scaled up and down according to the workload, in particular idling executors are removed and when there are tasks waiting to be executed, new executors are launched. Dynamic allocation can

be activate in Spark settings and should be used together with the External Shuffle Service, in this way data that have been manipulated from the executor is still available after the removal of the executor. Dynamic allocation has two different policy for scaling the executors:

- Scale Up Policy: new executor are requested when there are pending tasks, the number of executors is increased exponentially because they start slow and so the application might need a slightly higher number of them
- Scale Down Policy: idling executors are removed after a certain amount of time, this amount of time can be configured

In order for dynamic allocation to work, we must configure it, by setting the initial number of executor that are created when application starts and the minimum and maximum number of executor that can be reached when scaling down and up respectively. Dynamic allocation is available on all cluster manager currently supported by Spark, even in Standalone mode.

1.4.1 *Spark on Yarn*

Support for running Spark on YARN was added to Spark in version 0.6.0 and has been improved in subsequent releases [32].

When running on YARN, each Spark executor is run inside a YARN container. Spark supports two different modes to run on YARN, the *Yarn-cluster* and *Yarn-client* mode.

In client mode, as shown in Figure 1.9, the driver program is run inside the client process. In this way, the Application Master (AM) that is run in a YARN container is used only to request resources to the Resource Manager (RM). This mode is useful for interactive applications and for debugging purposes, since you can see applications' output immediately on the client side process. If the client disconnects from the cluster, the Spark application will terminate, this is due to the fact that the driver process resides on the client.

In cluster mode instead, as shown in Figure 1.10, Spark driver program is run inside the AM process managed by YARN. After initializing the application, client can disconnect from the cluster and reconnect later on. This mode makes sense when using Spark on YARN in production jobs.

Running on top of YARN cluster manager has some benefits. First of all YARN allows to dynamically share the cluster resources between the different frameworks that are running together. For example we can run MapReduce jobs after running Spark jobs without the need of changing YARN configurations. Moreover, YARN supports for categorizing, isolating and prioritizing workloads and employs security policies, in this way Spark can use secure authentication between its processes.

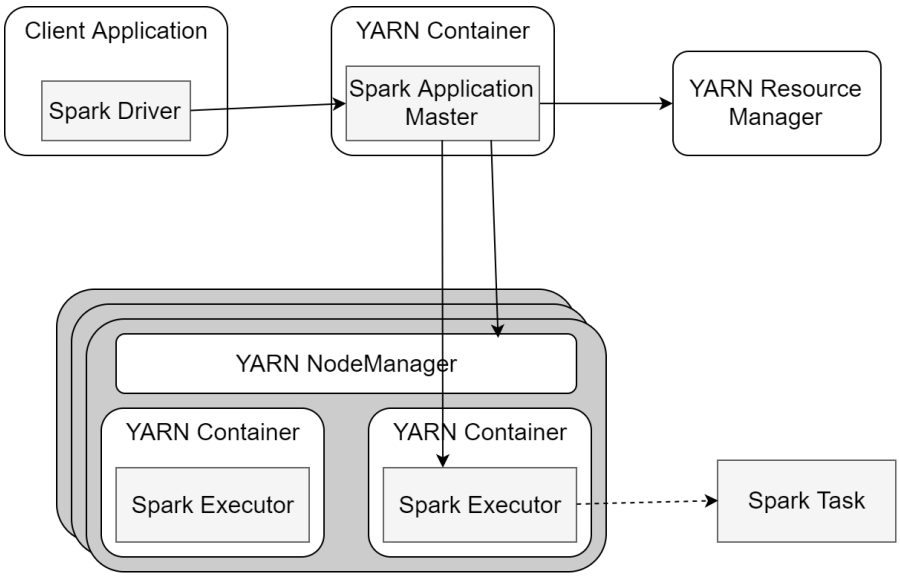


Figure 1.9: Spark on YARN Client Mode.

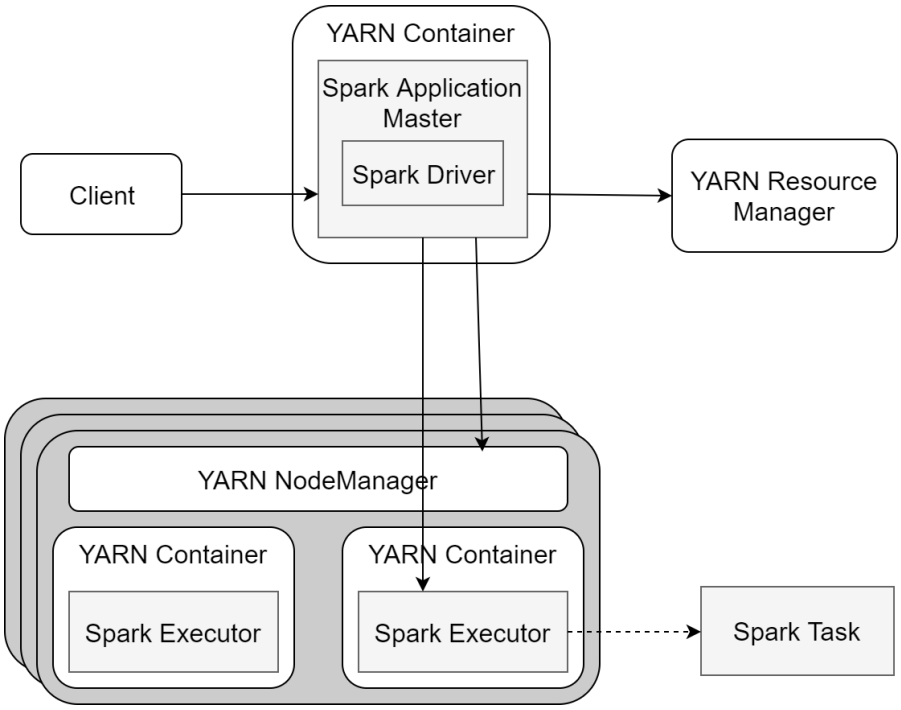


Figure 1.10: Spark on YARN Cluster Mode.

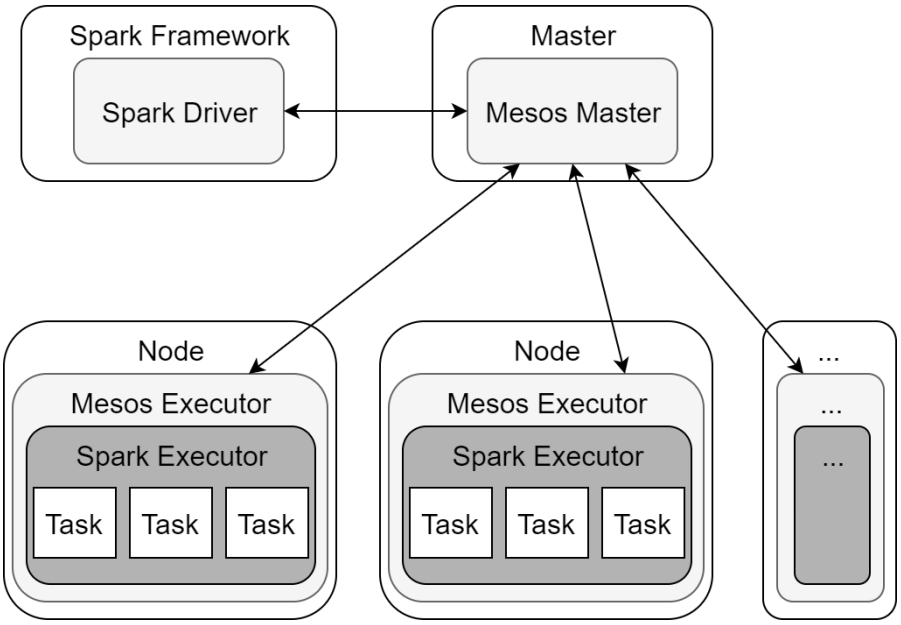


Figure 1.11: Spark on Mesos Coarse Grained Mode.

When running on YARN, Spark executors and driver program use about 6-10% more memory with respect to the standalone execution, this is due to the fact that this extra amount of off-heap memory is allocated in order to take into account YARN overheads.

1.4.2 *Spark on Mesos*

Support for running Spark on Mesos was added to Spark in version 1.5. Spark on Mesos can be executed in two different modes: coarse-grained and fine-grained [31].

In coarse-grained mode, as shown in Figure 1.11, each Spark application is submitted to Mesos master as a framework and Mesos slaves will run tasks for the Spark framework that are Spark executors. Mesos tasks are launched for each Spark executor and those Mesos tasks stay alive during the lifetime of the application unless we are using dynamic allocation or the executor is killed for various reasons. The advantage of coarse-grained mode is a much lower task startup overhead, with respect to the other mode, and so it is good for interactive session.

The drawback is that we are reserving Mesos resources for the complete duration of the application, unless dynamic allocation is active. Dynamic allocation allows to add and remove executors based on load: i) kill executor when they are idle, ii) add executors when tasks queue up in the scheduler. To use dynamic allocation it is required that the external shuffle service is running on each node.

In fine-grained mode, shown in Figure 1.12, Mesos tasks are launched for each Spark task, and those tasks die as soon as Spark tasks are

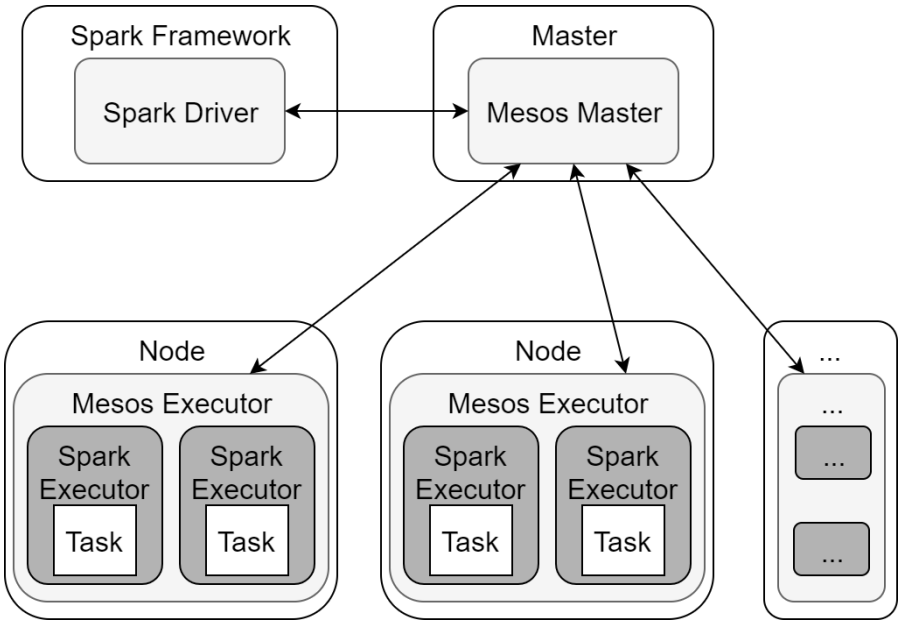


Figure 1.12: Spark on Mesos Fine Grained Mode.

done. This mode has too much overhead in case that Spark has too many tasks, for example if Spark application has 10,000 tasks, then Spark needs to be installed 10,000 times on Mesos agents. Because of this huge overhead, fine-grained mode has been deprecated since Spark version 2.0.0. This mode allows multiple instances of Spark to share cores at a very fine granularity, but it comes with an additional overhead in launching each task. Thus this mode is inappropriate for low-latency requirements like interactive queries or serving web requests, instead it is fine for batch and relatively static streaming.

Similarly to what happens on YARN, it is possible to run spark in *Mesos-client* or *Mesos-cluster* mode. In client mode, the driver process is executed in the client machine that submits the job, so it is required that it stays connected to the cluster for the entire time of the application execution. In cluster mode instead, the driver program is run on a machine of the cluster.

1.5 VIRTUALIZATION AND CONTAINERIZATION

Virtualization refers to creating the virtual version of something, included hardware components, storage devices and computer networks. Virtualization is born in 1960s, as a way to logically partition the system resources offered by a mainframe computer between the different application. From this point, the meaning of the word has been widely extended.

Virtualization is a technology that allows creating multiple simulated environments or dedicated resources from a single unique physical hardware system. An hypervisor is a software that can directly

connect to the hardware, with the purpose of splitting the unique physical system into more separate environment, each of them is different and secure, known as virtual machine (VM). These VM rely on the hypervisor ability of separating the hardware resources and distributing in a proper way.

The original physical machine equipped with the hypervisor is called *host*, meanwhile the VMs that are using its resources are called *guest*. These guest use the computation resources, such as CPU, memory and storage, as a set of resources that are easily re-allocable. The operators can control the virtual instances of CPU, memory, storage and other, such that the guests can receive all the resources they need in order to execute their operation. The words host and guest are used in order to distinguish the software that runs on the physical machine from the software that is running on the virtual one.

Hardware virtualization or platform virtualization refer to the creation of a VM that acts like a real computer with an OS. The software that is run in this VM is separated from the underlying hardware. This allows us to run particular configurations, for example we run a computer with a Windows OS that hosts a VM with Linux as guest OS.

There are at least two different hardware virtualization types:

- full virtualization: it completely simulates the hardware in order to allow the software, typically a guest OS, to be run without the need of modifications
- paravirtualization: the hardware environment is not simulated, anyway guest programs are run in isolated domain, as if they were run in completely separated systems. Guest programs need to be modified in a specified way in order to be run in this kind of environments.

In Figure 1.13, we can see the differences between the two kind of virtualization. In paravirtualization, the VM presents a different interface compared to the one of a physical machine. This requires modification in the guest OS in order to allow its execution inside the VM. The hypervisor exposes a set of APIs that the guest OS must use, in particular in order to execute privileged instructions. Calls to these particular functions are often defined as *Hypercall*. In full virtualization instead, VM have the same interface of the physical ones. Ideally, the guest OS could not be able to determine if it is being run on a physical or virtual machine. The great advantage of full virtualization is that we do not need to modify the OS, in this way the hypervisor can adopt a trap system in order to execute privileged instruction

We can improve the efficiency of the virtualization by using hardware-assisted virtualization, in particular we can decide to use CPUs that provide efficient support for virtualizing on hardware, but also other

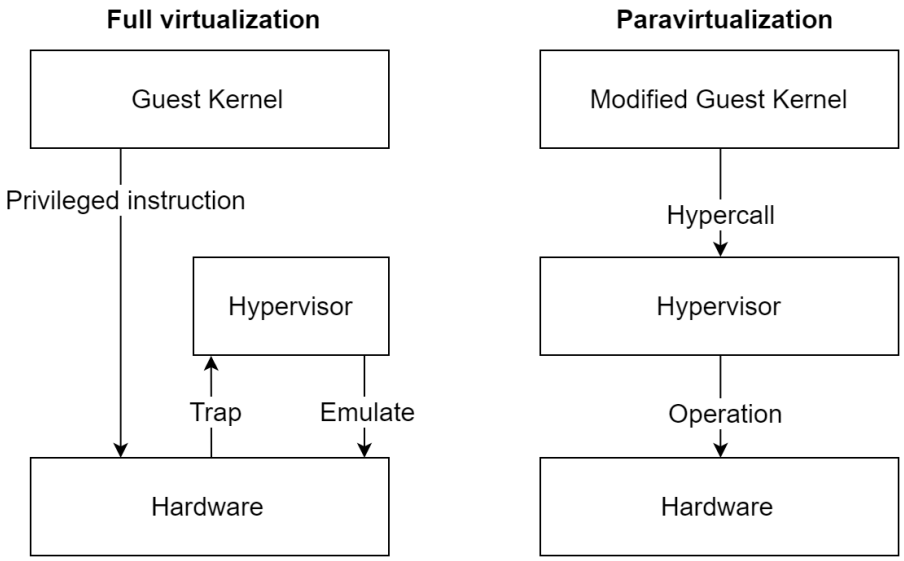


Figure 1.13: Full virtualization and paravirtualization.

kind of hardware components that can improve the performances of the guest environments.

Hardware virtualization can be seen as a trend of the enterprise IT that includes autonomic computing, a scenario in which the different environments are able to manage themselves based on th perceived activity, and utility computing, where the processing power is seen as a utility that users pay only when needed. The purpose of virtualization is to centralize the administrate task, offering scalability and good resource utilization. With virtualization, different OS can be run in parallel on a single CPU, this parallelism reduces overhead costs and is different from multitasking, where we only execute different programs in parallel on the same OS. Thanks to virtualization, an enterprise IT can better handle updates and rapid changes in OS and applications, with little impact on its users. Virtualization allows organizations to have better efficiency and availability of resources and applications.

It is important to remember that hardware emulation is a complete different thing from hardware virtualization, in particular with emulation we have a piece of hardware that imitates another piece of hardware. With virtualization instead, an hypervisor, which is a piece of software, imitates a piece of hardware or even an entire computer. Moreover, an hypervisor is not an emulator, even though both are software programs that imitate hardware, their domain of use is different.

Containerization is a OS-level virtualization technique that allows deploying and executing distributed applications without the need of launching an entire VM for each of the applications (Figure 1.14). These multiple isolated systems are called container, they are executed on top of a single host controller and the access a single kernel.

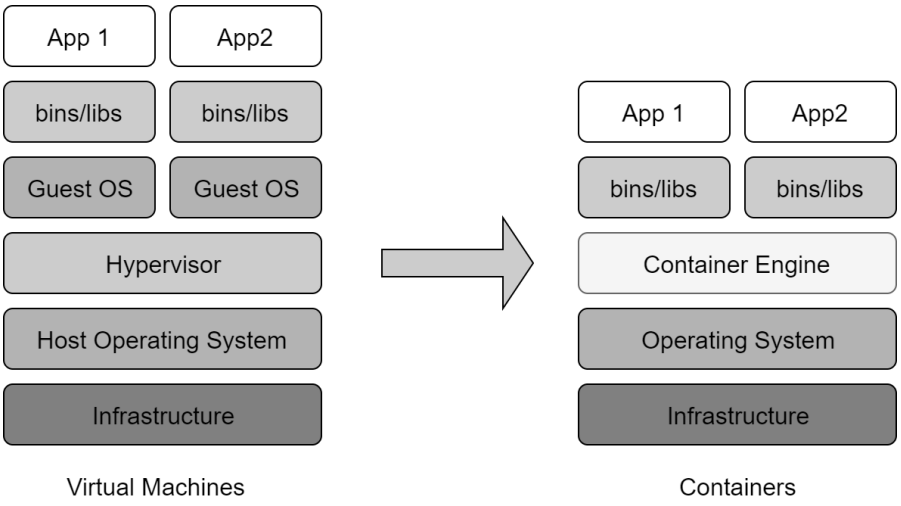


Figure 1.14: The difference in architecture between virtual machines and containers.

Since container share the same OS kernel of the host, they can be a lot more efficient than a VM, that instead needs a separate instance of the OS. Containers contain all the different components that are needed in order to execute the desired software, such as files, environment variables and libraries. The host OS controls the access of the container to the physical resources, such as CPU and memory, in order to prevent a single container from occupying the entire resources offered by the host.

The main advantages of containerization come from efficiency in terms of memory, CPU and storage, when compared to traditional hardware virtualization. Since containers do not have the same overhead of VM, in particular we do not need different instances of the OS, it is possible to support more containers in the same infrastructure. Containerization offers better performances since there is a single OS that takes care of all the hardware calls. A particular point of interest for the container is the fact that they can be create much faster than the instances that are based on an hypervisor, this allows to have a more agile environment and allows the creation of new approaches, such as the microservices and continuous integration and delivery ones.

Potential disadvantages of containerization might be the absence of isolation from the host OS. Since containers share the same host OS, a potential security threat might easily gain access to the entire system. This did not happen when using virtualization based on an hypervisor, since in this case the only compromised component would be the VM. In order to circumvent this problem, a solution might be creating containers inside an OS that is run from a VM, this prevents the security breach at container level from letting the attacker gain access to the OS of the physical host. Another little disadvantage of con-

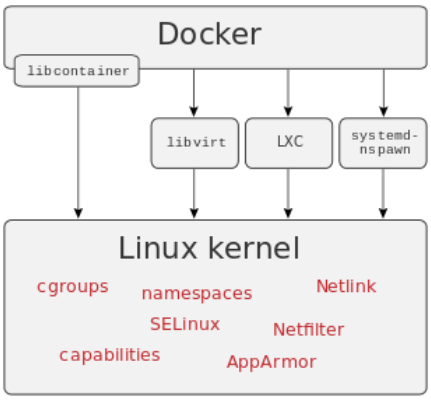


Figure 1.15: Docker can use different interfaces in order to access Linux kernel virtualization functionalities.

tainerization is that containers must execute the same OS as base OS, meanwhile instances based on an hypervisor are allowed to execute different OSs. Because of this, a container that is running on a Linux host, can neither execute an instance of Windows OS nor a Windows designed application.

Containerization has gained more and more relevance thanks to the diffusion of the open source software Docker, that has developed a way to give more portability to the containers, allowing them to be moved from different systems that share the same kind of host OS without the need for changing lines of code. In particular, with Docker container there are no environment variables that must be set on the guest OS or library dependencies that need to be managed.

1.5.1 *Docker*

Docker is an open source project that automatizes the deployment of applications inside software containers, giving a further abstraction thanks to OS level virtualization provided by Linux OS. Docker uses isolation functionalities provided by Linux kernel, such as *cgroups* and *namespaces* [8] in order to allow the coexistence of independent containers on the same Linux instance, avoiding the installation and maintenance of a VM.

Linux kernel *namespaces* isolate what the application ca see of the operating environment, including process tree, network, user ids and mounted file system. *Cgroups* instead provide resource isolation, including CPU, memory, I/O devices and network.

Docker implements a high level API in order to manage containers that execute in isolated environments. Since it uses Linux kernel functionalities, a Docker container, compared to a VM, does not includes a separated OS. Instead, it uses the kernel functionalities and exploits resource isolation and separated namespace in order to separate iso-

late what each application can see of the underlying OS. Docker can access Linux kernel virtualization functionalities using different ways, for example directly using `libcontainer` or indirectly using `libvirt`, Linux Containers (LXC) or `systemd-nspawn` (Figure 1.15).

Using containers, resources can be isolated, services can be limited and processes can be started in a way that each of them has a private perspective of the OS, with their own identifier, file system and network interface. More container can share the same kernel, but each of them can forced to use a different amount of resources such as CPU, memory and I/O.

By using Docker we can create and manage container in a way that simplifies the creation of distributed systems, allowing different applications and processes to work in an autonomous way on the same physical machine or on different virtual machines. This allows us to deploy new nodes only when necessary, in order to follow an evolution style that is similar to the platform as a service one.

2 PRELIMINARIES

In this chapter we introduce the previous work done in xSpark, on which this contribution is based on, and the related work.

2.1 XSPARK

xSpark is a Spark extension that offers optimized and elastic provisioning of resources in order to meet execution deadlines. This is obtained by using nested control loops. A centralized control loop, implemented on the master node, controls the execution of the different stages of an application; meanwhile multiple local loops, one per executor, focus on task execution.

In Figure 2.1 we can see a high level representation of xSpark's flow. A preliminary Profiling Phase, that is executed once per application, is composed by the execution of the application to obtain informations about the application, its logs are used to generate its Profiling Data.

During the Execution Phase, we control the execution of the application by means of xSpark's control loops. The centralized control loop is represented as a *Heuristic Based Planner*, which exploits the Profiling Data. The profiling data is used to understand the amount of work that is needed to complete the application's execution. This component uses the provided profiling data to determine the amount of resources to assign to executors during each of the application stages, in order to complete the execution before the provided deadline. The local loops instead are represented as *Control Theory Controllers*, they perform fine-grained tuning of the resources assigned to each of the executors using a control theory based controller. This component is used to counteract the possible imprecision of the estimated needed resources, which may be caused by different factors, such as the available memory, etc.

Usually Spark applications are not run only once, but they are long-lasting assets. As a consequence, xSpark can exploit an initial profiling execution to create an enriched DAG of the entire application, storing information about the various stages. For each stage, xSpark annotates the DAG with the execution time (stage duration), the number of task processed, the number of input records read, the number of output record written and the nominal rate, defined as the number of records that a single core processes during a second of execution. In Listing 2.1 we can see a portion, relative to stage number three, of the profiling data related to a PageRank application executed in xSpark.

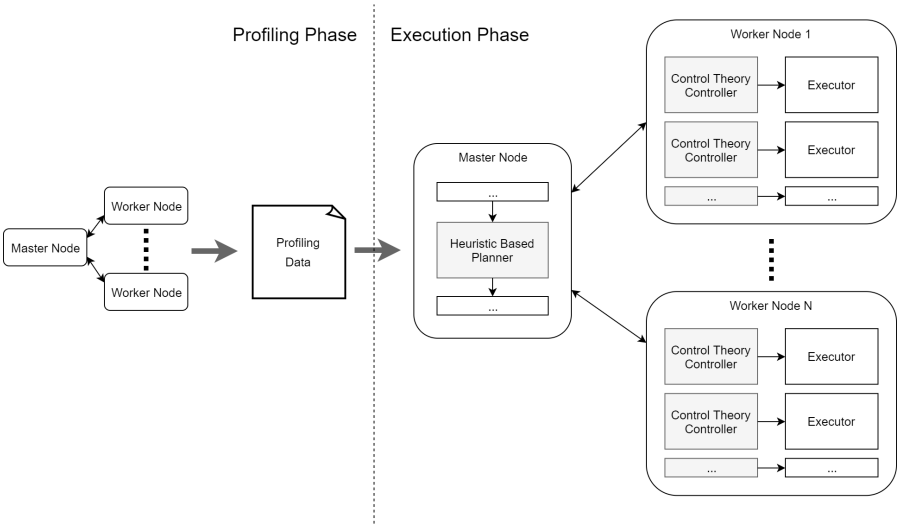


Figure 2.1: High level setup and execution flow of xSpark.

For example, in *duration* field there is the serialized total duration in milliseconds of the tasks.

Listing 2.1: Example of profiling data from a PageRank application

```
{ ...
   "3": {
                ...
                "cachedRDDs": [16, 9],
                "duration": 504086,
                "genstage": false,
                "name": "mapPartitions at ...",
                "nominalrate": 413525.2675138766,
                "numtask": 1000,
                "parentsIds": [1, 2],
                "recordsread": 1000,
                "recordswrite": 0.0,
                "shufflerecordsread": 208452298,
                "shufflerecordswrite": 1000000,
                "skipped": false,
                "weight": 4.97875957673889
                ...
        },
... }
```

At runtime, the annotated DAG allows us to comprehend how much work has already been completed and how much work still needs to be done. This means that xSpark can only optimize the allocation of the resources if all the executions of the same application use the same DAG. This might not always be the case, for example when the code contains branches or loops, because these might need to be resolved in different ways at runtime. If the application DAG at runtime

differs from the one obtained during the profiling phase, xSpark is not able to estimate the amount of remaining work.

The centralized control loop is activated before the execution of every stage, it uses an heuristic, explained in Section 2.1.2, in order to assign a deadline to the stage, calculate the amount of CPU cores that are needed to satisfy it, and assign cores to the allocated executors. The per-stage deadline takes into account the amount of work already completed, the consumed time, and the overall deadline. All the computations done by the heuristic are based on the information stored in the DAG and obtained during the profiling phase. Unfortunately many factors can influence the actual performance and invalidate the prediction, such as the amount of records that have been filtered-out, the available memory, the number of used nodes, the storage layer dimension, and so on. It is important to remember that Spark mostly uses in-memory data, but there are operations like `textFile`, `saveAsTextFile` and `saveAsSequenceFile` that impose restricting constraints on the storage layer. If not correctly dimensioned, the storage layer might become a bottleneck, causing the throughput to degrade and thus making the provisioning predicted by xSpark incorrect.

Local control loops, explained in Section 2.1.3, counteract this imprecision by dynamically modifying the amount of CPU cores assigned to the executors during the execution of a stage. This can lead the executor to use more or less resources than the ones previously assigned by the centralized control loop. The local loop controls the progress of a specific executor with respect to the tasks it has assigned. A control theory algorithm determines the amount of CPU cores that must be allocated to the executor for the next control period, typically one second, and assigns them.

xSpark uses Docker in order to dynamically allocate CPU cores and memory, as explained in Section 2.1.1. Memory allocation simply sets an upper bound to the memory that each docker container (executor) can use. CPU cores instead are allocated in a more sophisticated way. Using Linux cgroups, Docker can support CPU shares, reservations and quotas. In particular, xSpark uses CPU quotas. CPU shares are not able to limit the number of CPU cores used by a container in a deterministic way, in particular it is not independent of other processes running on the same machine. CPU reservation instead does not have the fine granularity we are looking for, indeed the allocation would be limited to entire cores. By using CPU quotas instead, we have a reliable and tunable mechanism that provides also fine granularity allocation, in particular it allows xSpark to allocate fractions of cores to the containers (executors), with a precision up to 0.05 cores.

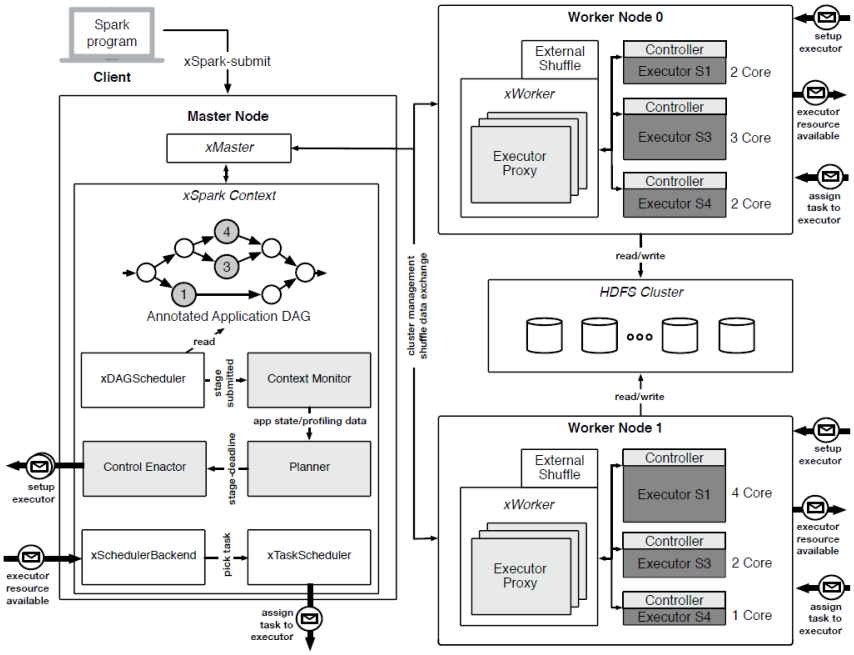


Figure 2.2: Architecture of xSpark. New components are represented in light grey boxes, meanwhile those that start with an *x* are the modified ones. In dark grey there are the containerized components (the executors).

2.1.1 *Architecture*

To achieve the objectives of xSpark, the architecture and processing model of Spark have been modified. In Figure 2.2 we can se how xSpark architecture differs from Spark one.

The principal architectural change introduced by xSpark is an increased focus on stages. Instead of considering entire applications, xSpark reasons on per-stage deadlines. xSpark instantiates an executor per stage per worker, instead of a single executor per worker for all the stages that will be executed. This way the resources that are allocated to a single executor only impact the performance of the stages that are associated with it, this leads to a fine grained control over the different stage, and thus on the entire application. When multiple stages are run in parallel, multiple executors can be running on the same worker node. When a stage is submitted for execution, one executor per worker is created and bound to that stage. This way computation and data are equally spread across the entire cluster. Thanks to containers, xSpark can isolate the execution of the different executors that are running on the same worker node and achieve quick, fine-grained resource provisioning. On average, containers can be modified in less than one second, allowing a more precise way of allocating cores.

Users submit applications and their deadlines to the master node using the `submit` command. This creates a *xSparkContext* on the master node, containing a *xMaster* object that is used to manage the cluster and that knows all the resources that are available on each worker node. A *xSparkContext* is composed by six components:

- *xDagScheduler* schedules the stages according to the application's DAG. The submitted stage is enriched with the informations obtained during the profiling phase.
- *ContextMonitor* monitors the progress of the application, taking into account stage scheduling and completion. It also stores information about the performances of the execution, that will be later used to calculate the deadlines and the resources needed by upcoming stages.
- *Planner* is heuristics based and is used to calculate the deadlines and resources associated with a stage.
- *ControlEnactor* determines when a stage is ready to be executed, meaning there are enough resources (cores) and sufficient executors become available. It also has the job of initializing the different executors.
- *xSchedulerBackend* controls the stage execution. In particular it launches new tasks by taking into account the resources availability and registers their completion.
- *xTaskScheduler* also controls stage execution, with the goal of allocating tasks to the available cores to optimize data locality. In general, the closer a data partition required by a task is to the task's executor, the better.

xSpark also modified the worker nodes. Each node contains a *xWorker* that connects to a *xMaster*, generates local controllers for the executors, and controls the evolution of its executor by dynamically scaling their resources. *xWorker* creates an *Executor Proxy* for each of the executors that are running on its node. These proxies are placed between the executors and the *xSchedulerBackend*, and are used to monitor the execution progress of the stage assigned to the executors. It is important to remember that each executor focuses on a single stage at a time. The heuristic calculates how many tasks must be executed by each of the executors of the same stage, the strategy is to have en equal number of tasks assigned to each of the executors. This way the deadline assigned to each of the executors coincides with the deadline of their stage. This allows the different executors of the same stage to not be synchronized. Native Spark instead requires that executors with free resources spontaneously require new tasks to execute from the master node.

Every *xWorker* uses an *External Shuffle Service*. Native Spark moves data across the cluster in different ways. If the data is stored on executor's memory, then the executor itself manages the data exchange. If the data is stored on an external storage system (e.g., HDFS cluster), they can be retrieved by using different communication protocols. If the data is stored in the internal storage of a worker node, then the data is managed by the *External Shuffle Service*. Notice that this is not the default, but xSpark adopts this technique to be able to assign zero CPU cores to an executor, without loosing the ability to read data, since it is effectively not performed by the executor but by the external service.

2.1.2 *Heuristic*

xSpark uses a heuristic to compute per-stage deadlines and to estimate how many cores must be allocated for a stage to successfully fulfill the deadline. In order to do this, at submission time the user is asked to specify three parameters: i) the application deadline, ii) the cluster size, and iii) the number of cores per worker node. Before executing the application, xSpark performs a feasibility check given the available resources.

When a stage is submitted for execution, its deadline is computed

$$deadline(s_k) = \frac{\alpha \cdot ApplicationDeadline - SpentTime}{weight(s_k)}$$

where $SpentTime$ is the time already spent for execution and α a value between 0 and 1 that xSpark uses to be more conservative with respect to the provided $ApplicationDeadline$. The weight is computed as

$$\begin{cases} w_1(s_k) = \#(RemainingStages + 1) \\ w_2(s_k) = \frac{\sum_{i=k}^{k+w_1} duration(s_i)}{duration(s_k)} \\ weight(s_k) = \beta \cdot w_1(s_k) + (1 - \beta) \cdot w_2(s_k) \end{cases}$$

where w_1 is the number of stages still to be scheduled (s included) and w_2 is the rate between the duration of s and the duration of the remaining stages (s included).

xSpark then proceeds to estimate how many cores are needed to execute the stage:

$$estimatedCores(s_k) = \lceil \frac{inputRecords(s_k)}{deadline(s_k) \cdot nominalRate(s_k)} \rceil$$

where $inputRecords$ is the number of records that will be processed by s_k and $nominalRate$ is the number of records processed by a single core per second in stage s_k.

Since xSpark controls the resource allocation of a stage both before and during the execution, the maximum amount of allocable cores needs to be greater than the estimated one, in order to be able to accelerate when progressing slower than expected

$$maxAllocableCores(s_k) = overscale \cdot estimatedCores(s_k)$$

The final step is to determine the initial number of cores that should be assigned to the different executors, xSpark distributes the cores equally amongst the available workers by creating one executor per stage per worker. In this way, it is guaranteed that executor performances will be equal, and that xSpark can compute the same deadline for all the executors. The initial number of cores per executor is computed as

$$initCorePerExec(s_k) = \lceil \frac{maxAllocableCores(s_k)}{overscale \cdot cq \cdot numExecutors} \rceil \cdot cq$$

where $numExecutors$ is the number of executors and cq is the *core quantum*, a constant that defines the quantization applied to resource allocation, the smaller this value is, the more precise the allocation is.

2.1.3 *Controller*

Each containerized executor has an associated local controller, whose goal is to fulfill the per-stage deadline taking into account external disturbances by dynamically allocating CPU cores. The controllers use control theory, with no heuristic involved.

The centralized control loop determines the desired stage duration, the maximum and the initial number of cores that should be assigned to the executors and the number of tasks that must be processed. Local controllers adjust the number of allocated cores, according to the work that has already been accomplished.

Executors that are dedicated to different stages are implicitly independent, and thus their controllers are also independent. The executors that are running in parallel on the same stage must complete the same amount of work (number of tasks) in the same desired time. This means that local controllers are independent and do not need to communicate amongst themselves. Moreover, the heuristic is relegated outside the local controller, thus it cannot compromise the controller's stability.

In the controller the progress set point is chosen based on the desired completion time, this value is received from the centralized control loop. In Figure 2.3 we see the prescribed completion percentage, in particular t_{co} is the desired completion time and $\alpha \in (0, 1]$ is a configuration parameter used to determine earlier with respect to deadline, we are willing to complete the execution. In order to track the set point ramp, we need to use a Proportional plus Integral (PI) controller. As a result, the discrete-time controller in state space form reads:

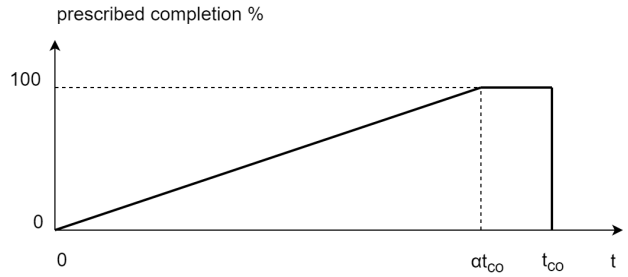


Figure 2.3: Set point generation for an executor controller.

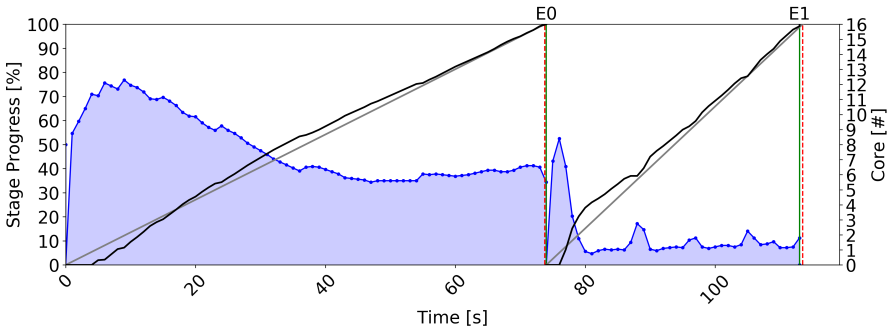


Figure 2.4: CPU cores allocated to the application *aggregate-by-key* running on a xSpark worker node. The blue line represents the allocated cores over the time, gray and black line are desired and actual progress rate respectively. Green line represents the obtained stage ending, meanwhile dashed red one is the desired one.

$$\begin{cases} x_C(k) = x_C(k-1) + (1-\alpha)(a^o_{\%}(k-1) - a_{\%}(k-1)) \\ c(k) = K x_C(k) + K(a^o_{\%}(k) - a_{\%}(k)) \end{cases}$$

where $a^o_{\%}$ is the prescribed progress percentage at each k control step and $a_{\%}$ is the accomplished completion percent at each k control step. Notice that it is possible that the controller computes a negative value for $c(k)$, the CPU cores that need to be allocated. To fix this problem $c(k)$ must be clamped between a minimum c_{min} and a maximum c_{max}. To maintain consistency, we need to recompute the state $x_C(k)$ as

$$x_C(k) = \frac{c(k)}{K} - a^o_{\%}(k) + a_{\%}(k)$$

In Figure 2.4 the cores allocated to an application executor running on a worker node are shown. The running application is *aggregate-by-key* and is composed by two stages. The black line represents the progress of the stage, meanwhile the gray one is the desired progress rate. The goal of the controller is to reduce the error, i.e., the distance between the two lines. We want the black line to follow as best as possible the gray one.

2.2 RELATED WORK

Spark offers the feature of adjusting the quantity of resources used by an application in a dynamic way. Executors are requested when a task remain pending for a certain amount of time (user defined). This action is applied more times if the queue of pending tasks is still non-empty. On the other hand, an executor is removed if it remains idle for a too long time. Anyway, this dynamic allocation is limited to the executor granularity, since it is only possible to add or remove executors with a fixed amount of CPU cores. This is way far from xSpark granularity. xSpark works in *Spark Standalone Mode*, where master and worker node are launched by means of scripts. It is possible to manage Spark clusters using Yarn or Mesos, both of them help in scheduling and allocating resources in the cluster, but none of them focuses on allocating resources according to application-level QoS.

In academic literature, using predefined deadlines when engineering Spark application is still under investigation. Gibilisco et al. [11] propose a performance model for DAG-based applications, allowing to have an accurate prediction of how the application will behave according to a specific input data set size and configuration settings, without taking into account dynamic resource allocation.

Lots of work instead has been done on engineering techniques for scheduling MapReduce applications taking into account deadlines and considering relevant aspects such as fairness, fault-tolerance, user priorities and data locality. Chen, Lin, and Kuo [4] propose a new MapReduce scheduler based on bipartite graph modeling, in order to obtain optimal solution of the deadline-constrained scheduling problem, as the job progresses, the scheduler can dynamically find different computing resources for running the job without violating the job deadline. He, Lu, and Swanson [15] propose a new real-time scheduler for MapReduce, that avoids accepting jobs that will lead to deadline misses and improves cluster utilization. Teng et al. [35] propose an algorithm for scheduling hard real-time tasks on a MapReduce-based cloud, providing a theoretical analysis of the scheduling performances and proving a bound on cluster utilization, which can be used in order to determine if a given task set can be scheduled. Wolf et al. [40] propose a flexible scheduling allocation scheme, trying to optimize a variety of standard scheduling theory metrics (e.g., response time) while ensuring the same minimum and maximum job slot guarantees as in Hadoop Fair Scheduler. Zaharia et al. [41] propose a simple algorithm to address the conflict between locality and fairness, when a job that should be scheduled next according to fairness cannot launch a local task, it waits a small amount of time letting other jobs launch tasks instead, in this way near optimal data locality is achieved.

Concerning scheduling techniques that take into account some form of dynamic allocation, notable works have been proposed. Polo et al. [28] propose a new tasks scheduler for a MapReduce framework, that allows performance-driven management of MapReduce tasks, the scheduler dynamically predicts the performances of concurrent MapReduce jobs and adjusts the resource allocation for the jobs, this allows meeting application performance objectives without over provisioning physical resources. Verma et al. [38] propose the ARIA framework, composed by three inter-related components: i) a job profile that summarizes critical performances characteristics of an application, ii) a MapReduce model that for a given job (with a known profile) and its soft deadline, estimates the amount of resources required to job completion within the deadline, iii) a soft deadline based scheduler, that determines job ordering and resources needed to be allocated. Lama and Zhou [21] propose Aroma, a system that automates the allocation of heterogeneous Cloud resources and configuration of Hadoop parameters, in order to achieve QoS goals while minimizing the incurred cost, it addresses the challenge of provisioning ad-hoc jobs that have performance deadlines trough a two phase machine learning and optimization framework.

There is more relevant work on scheduling techniques that do not focus on dynamic resource allocation. Phan et al. [27] provide a case study of the online scheduling of MapReduce jobs executed by Hadoop, showing that the existing Hadoop scheduler is ill-quipped to handle jobs with deadlines. By adapting existing multiprocessor scheduling techniques for the cloud environment, it has been observed a significant improvement in minimizing missed deadlines and tardiness. Kc and Anyanwu [19] extend real time cluster scheduling approach in order to take into account the two-phase computation style of MapReduce, developing criteria for scheduling jobs based on user specified deadline constraints and evaluating a scheduler that ensures that only jobs whose deadlines can be met are scheduled for execution.

3 SOLUTION

Our goal is to support the coexistence of multiple Spark applications running on the same infrastructure each with its own controller that tries to follow the desired progress rate.

One of the ideas we had was to speed up stage computation when the system load is not high, by allocating more resources than the ones strictly needed to satisfy the desired progress rate. This way when a new application is submitted, and the system load becomes high, we can make easily tolerate disturbances.

A problem that can result from this proposed solution is that, if there is a single application in the system, it will finish the computation earlier than the desired deadline. This problem has been addressed by providing a way to enable or disable this speed up mechanism.

We also tackled the problem of how to partition the resources among the different applications, since the system resources might not be enough to satisfy all the application requests. Different ways of managing resource contention are discussed.

Concerning the memory of the application, we choose for a static allocation of heap memory and a dynamic allocation of off-heap memory. In particular we decided to assign a small portion of heap memory to each of the applications and to split the remaining portion as off-heap memory between all the applications that are using off-heap. Since it was not previously supported, we developed a way to resize the assigned off-heap memory to an executor.

3.1 CHANGES IN THE ARCHITECTURE

Not many changes to xSpark's architecture were requested to support multiple applications. Since the only point of contact between the applications is the allocation of cores and memory, we were able to preserve most of xSpark architecture.

xSpark instantiates one executor per stage per worker node per application. This way the dynamic resources allocated to a single executor will only impact the performances of the stage and of its associated application. This allows us to perform a fine-grained control over the different stages of the different applications. Executor's memory manager has been modified in order to support the resizing of the off-heap maximum allocation memory (Section 3.4).

Concerning *xMaster*, we needed to modify the previous heuristic (Section 3.2) to support applications that are running slower than ex-

pected, due to disturbances provoked by the presence of other applications running in parallel. Furthermore, we needed to support the instantiation of multiple executors per worker that have access to the entire set of available cores (Section 3.3). The reason is that we will handle the allocation of cores to the different applications running.

On the *xWorker* hand, each of the executors will have its own *Executor Proxy* and *Controller*. In particular, the *Controller* structure has been extended (Section 3.5) in order to be aware of the presence of other applications and take care of them. We moved to a hierarchical structure in which there is an entity that supervises all the controllers running on the *xWorker* and adjusts their behavior. The main purpose of this entity is to handle the situation in which the different applications try to allocate an amount of resources that is larger than the ones available on the *xWorker* (Section 3.6).

3.2 HEURISTIC

As explained in section 2.1.2, the centralized control loop of xSpark applications works by means of heuristics. This heuristic has the purpose of calculating the per-stage deadline and the amount of CPU cores that are needed to complete the execution before the deadline.

Since the heuristic works independently for each application, it doesn't know the complete state of the system. In particular, it does not know if there are other applications running that might slow down the execution of the currently analyzed application. The problem is that, in order to keep up with the desired progress rate, the heuristic might suggest a *maxAllocableCores* value that is larger than the number of CPU cores actually available in the system. This problem has been solved by keeping its value under the maximum available system resources,

$$\begin{aligned} &\mathtt{maxAllocableCores}(s_k) = \\ &\qquad \min(\mathtt{overscale} \cdot \mathtt{estimatedCores}(s_k), \mathtt{TotalCores}) \end{aligned}$$

where *TotalCores* is the number of CPU cores that are usable in the cluster, defined by the product of the number of running executors and their available cores. This way execution can continue, even though it may no longer be possible to complete the execution before the desired deadline.

3.3 LAUNCHING APPLICATIONS

When launching an application, xSpark waits until the number of executors launched for that application, that are alive, reaches what is specified in the configuration parameter `spark.control.maxexecutor`.

An application can specify the total number of cores that its executor should have assigned, by default this is set to unlimited. This

way, in native Spark in Standalone mode, the first application that is launched will acquire all the resources available and launch its Spark executors. The next application will need to wait until the previous one completes. Spark keeps track of the available resources in the workers in terms of free CPU cores and memory.

To launch multiple applications we bypass the check on available cores and allow all the applications to start, given there is enough free memory to create a JVM process for the executor. We then can dynamically tune the number of cores assigned to each executor, since they are run inside containers. We cannot do the same thing on the JVM heap memory, because its size is determined before it is created. Launching executors without taking into account memory consumption often results in the executors crashing because of the impossibility for the JVM heaps to grow.

Concerning the cores, xSpark's implementation guarantees that, at any given time, the total number of cores assigned to the executors of a given worker is lower or equal to the worker's available ones.

$$\sum_{i}^{I} cores_i \leqslant MaxCores, \quad \forall\, worker$$

where $cores_i$ is the number of cores associated to the executor $i \in I$, the set of all executors of a given worker. The value of $MaxCores$ is determined by the configuration parameter `spark.control.coreforvm`.

3.4 SCALABLE OFF-HEAP MEMORY

One of the main problems when executing a Spark job is how to determine the amount of memory to allocate to each executor. Spark allows us specifying this value by tuning the parameter `spark.executor.memory` in an application's properties [33]. This will change the size of the executor's process memory heap. The same configuration can be applied to the application's driver process by editing the value of `spark.driver.memory`. These settings change the value of the maximum heap size in the JVM process, which is defined with the parameter `-Xmxn` when launching a Java application. The n in `-Xmxn` specifies the maximum size of the memory allocation pool [18].

When dealing with a single application, choosing the values is simple: we just need to pay attention to the system's available memory and pick a value that is smaller than the system's total memory. In general, picking a larger value will improve the performances of the application by reducing the probability of disk swap, which is a serious threat because it will inevitably degrade application efficiency. Still, we need to pay attention to the system's load, because when the heap tries to grow, and there is not enough free memory in the system, the process will terminate and the JVM will be restarted.

HEAP SIZE	NUMBER OF APPLICATIONS
100 GB	1
50 GB	2
25 GB	4
10 GB	10

Table 3.1: Example of how changing the heap size of the executor's and driver's process determines the number of applications that can run in parallel. The total system memory is 100 GB.

Choosing the right value for the heap processes in a multi-application context adds another point to the problem. In general, we do not know how many applications will be submitted to our Spark cluster. If this number were known, we could simply equally partition the available memory to the applications, keeping in mind that we want to keep the total amount of JVM heap sizes below the system maximum memory.

$$Heap_i = \frac{SystemMemory}{|I|}$$

where I is the set of applications running, $Heap_i$ is the amount of heap memory allocated to the application $i \in I$, and $SystemMemory$ is the total memory available. Instead, when this number is unknown, we have to keep in mind that changing the heap size will result in changing the maximum number of applications that can be run in parallel, as we can see in Table 3.1.

Spark instantiates one *Memory Manager* for each JVM. It enforces how memory is shared between execution and storage. In this context, *Execution Memory* refers to the part of memory used for computation in shuffles, joins, sorts and aggregations, while *Storage Memory* refers to the part used for caching and propagating internal data across the cluster.

Starting with Spark 1.6, the default memory manager is *Unified Memory Manager* and memory is divided in 3 regions (Figure 3.1) [12]. *Reserved Memory* is the memory reserved by the system and its size is hardcoded. Even though it is called reserved, it is not used by Spark in any way but instead it sets a limit on what you can allocate for Spark usage. *User Memory* is the memory pool used by the user, for example to store his/her own data structures. *Spark Memory* is the memory pool managed by Spark, the pool is split into two regions: storage memory and execution memory. The size of User and Spark Memory are set by `spark.memory.fraction` value, while the sizes of Storage Memory and Execution Memory are determined by `spark.memory.storageFraction`. The advantage of this memory man-

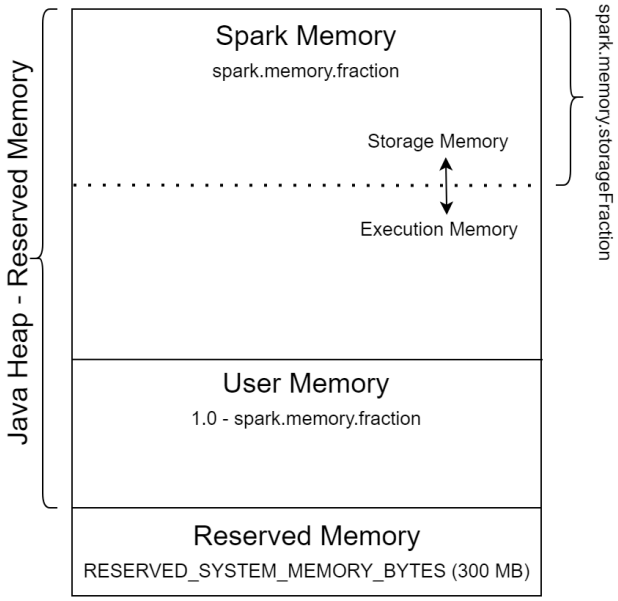


Figure 3.1: Spark Unified Memory Manager introduced in v1.6.

agement scheme is that the boundary is not static, and can therefore be moved in case of pressure.

The main problem when dealing with heap sizes is that it is not possible to resize JVM heap memory at runtime, this can only be done by killing the process and then restarting with other parameters. This might be a problem when deciding how much memory to allocate to each application. Knowing that Spark will postpone the launch of an application if the requested allocation of memory is not satisfiable, we need to pay attention when choosing the application memory value. Assigning a high amount of memory will cause application executions to be serialized, while deciding for a lower value will allow us to run a higher number of applications in parallel at the price of an increased risk of disk swapping.

A possible workaround is to use off-heap memory to add flexibility in terms of memory boundaries. Even though the best performance is obtained when operating only in on-heap memory, Spark can use off-heap allocation both for execution and storage memory. In Section 5.5 we will compare the performances of running applications with and without off-heap allocation. Off-heap refers to objects managed directly by the operating system and stored outside the process heap, thus not processed by the garbage collector. Accessing data off-heap is slightly slower than accessing data on-heap, but it is faster than reading and writing from a disk [20].

Spark provides a way to configure the amount of memory to be used off-heap. `spark.memory.offHeap.enabled` should be set to `true` and `spark.memory.offHeap.size` defines the total amount of memory in bytes for off-heap allocation (strictly larger than 0). Notice that it has no impact on heap memory.

As with heap memory, Spark does not provide a way to resize the memory to be used by off-heap objects at runtime. Luckily, since off-heap memory does not reside inside a JVM process, a simple resizing mechanism can be implemented thst reduces or increments off-heap storage and execution pool sizes when needed.

This feature has been achieved by adding to *Unified Memory Manager* a way to resize the execution and storage pools by specifying the new total size of the usable off-heap memory. Spark already provides a way to free memory by writing blocks to disk. This implementation of Spark *Memory Manager* is very flexible, since we do not have hard boundaries between storage and execution memory pools. The storage pool can borrow as much free execution memory, until the execution pool reclaims its space. When this happens, cached blocks will be evicted from memory until sufficient memory is released to satisfy execution memory request. Thanks to this it has been possible to reduce, in an unbalanced way, the two pools, by only evicting storage pool blocks if necessary, which resulted in an easier implementation. Executors are informed by their worker when they are asked to resize their off-heap memory, this happens when a new application is starting and when one terminates.

With this feature, we can decide to launch an application with a relatively small heap memory size and dynamically resize its off-heap memory according to the number of applications that are running at a given time. The off-heap memory that is usable by an application is calculated as

$$\mathrm{OffHeap} = \frac{\mathrm{SystemMemory} - \sum_i^I \mathrm{Heap}_i}{|I|}$$

where I is the set of applications running, Heap_i is the amount of heap memory allocated to the application $i \in I$ and $\mathrm{SystemMemory}$ is the total memory available.

3.5 CONTROLLER

As previously introduced in Section 3.1, to support multiple applications running together on the same worker, we need to be sure that the total resources allocated to the applications do not exceed the resources that the worker is offering. This is an important point, since allocating a total amount of resources to the containers that is greater than the ones available in the system leads to resource contention.

In the original implementation of xSpark, every executor controller calculates the optimal CPU cores allocation for that executor, so that it could follow the desired progress rate, as discussed in Section 2.1.3. Now, we want to have a hierarchical structure in which a per worker supervisor controls the behavior of the controller executors running. It should remember that on each worker there is one controller executor per running application, since only one executor per application

is launched on every worker. The idea is that every controller executor still proposes a desired CPU cores value in order to follow the progress rate, but the supervisor can decide to modify this value according to the resource situation.

Algorithm 1 shows how the supervisor collects all the CPU cores requested by the executor controllers to be able to follow their desired progress rate. It simply computes the value of CS_{RATE} for each of the running executors. The procedure that calculates the value is the same as the one used in xSpark. At the end of the procedure, CS_{RATE} contains the requested cores by all the executors.

Algorithm 1 PI Controller that generates the core allocation in order to follow the desired rate

for $i = 1$ **to** N **do**
  if $PV[i] < 1.0$ **then**
    $CSP[i] = K * (SP[i] - PV[i])$
    $CSI[i] = CSI_{OLD}[i] + K * \frac{T_s}{T_i} * (SP[i] - PV[i])$
    $CS_{RATE}[i] = max(0, CSP[i] + CSI[i])$
  else
    $CS_{RATE}[i] = 0$
  end if
end for

Given the previously calculated CS_{RATE}, the supervisor computes a different core allocation, with the purpose of distributing all the cores available in the worker node to the executor. From Algorithm 2 we can notice that this operation is performed only if any of the executors still is requesting resources, which means that it has not yet finished its assigned task. The $Zeroes(N)$ procedure generates a zero cores allocation for the N executors, while $DistributeCores(CS_{RATE})$ allocates the system available cores according to different strategies that will be discussed in Section 3.6. The allocation is called CS_{ALL} because its purpose is to saturate the resources of the worker node.

Algorithm 2 Generation of the core allocation in order to distribute the entire system resources

$SUM\_CS_{RATE} = \sum_i CS_{RATE}[i]$
if $SUM\_CS_{RATE} = 0$ **then**
  $CS_{ALL} = Zeroes(N)$
else
  $CS_{ALL} = DistributeCores(CS_{RATE})$
end if

Once both allocations CS_{RATE} and CS_{ALL} have been calculated, the supervisor needs to check the situation of the node. In particular what we want to check is the presence of a resource contention state.

When executing multiple applications in parallel, each of them will request a certain amount of resources (CPU cores) in order to keep up with the desired progress rate. An application is unaware of other applications, so it is possible that it will ask to allocate all the system resources. According to the requests of all the applications, we can identify two possible situations. We call rc_i the cores requested by application $i \in I$.

If the total request is less than or equal to the available resources of the system (max_cores),

$$\sum_{i}^{I} rc_i \leqslant max\_cores,$$

each of the applications will obtain the allocation of the requested resources. This way, the applications do not notice the presence of others and can continue their execution as if they were the only one present in the system. In this case we say that there is no contention between the different running applications.

Instead, if the total request is higher than the available resources of the system,

$$\sum_{i}^{I} rc_i > max\_cores,$$

we enter in a state of resource contention. Contention means that the requests of the different applications cannot be satisfied, because there are not enough resource to fulfill all the allocations. What to do in this case is to correct the amount of resources to try to restore a state of absence of contention.

This situation is easily visualizable with two applications in a two dimensional plan, as shown in Figure 3.2, where on the two axes we have the resources requested by two different applications. The maximum amount of resources allocable is fixed at 8 CPU cores. The white point represents a feasible allocation, because the sum of the requested cores ($3 + 2 = 5$) is lower than 8. Instead the black point represents an infeasible allocation, because the sum ($4 + 7 = 11$) is greater than 8, thus we are in a state of contention. The region above the dashed line represents all the possible resource requests that will cause contention. When

$$rc_{app1} + rc_{app_2} \leqslant max\_cores$$

we are in a situation where there is no contention, instead when

$$rc_{app1} + rc_{app_2} > max\_cores$$

we are in a contention state. In case of contention, we need to find a new pair cc_{app1} and cc_{app2} such that $cc_{app1} + cc_{app2} \leqslant max\_cores$, where cc_i stands for corrected cores of the application i.

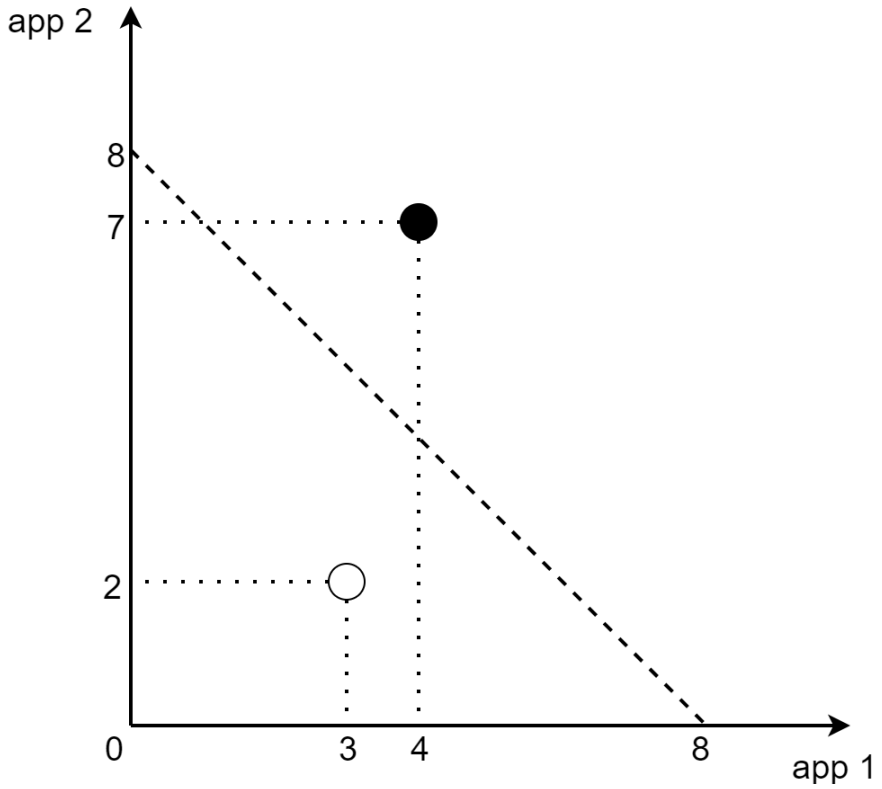


Figure 3.2: Example of two possible resource requests by two applications running in a system where the maximum number of allocable cores is 8.

Resource contention is checked by comparing the sum of the controllers CS_{RATE} values with the total number of available cores, CS_{MAX}. Once we know the state of the system, as we can see in Algorithm 3, we can determine what to do. In particular, if there is a state of contention, i.e. $\sum_i CS_{RATE}[i] > CS_{MAX}$, our only possible choice is to allocate the cores stored in CS_{ALL}, since this will guarantee that the total number of allocated cores is equal to the ones of the system. If there is no contention, we can use the configuration variable γ to move with continuity between two different behaviors of the system. In particular, with $\gamma = 1$ we decide to saturate the system resources, thus applying the CPU cores defined in CS_{ALL}. This way we achieve the complete utilization of the resources. Using $\gamma = 0$ we decide to only allocate the resources that have been requested by the executors' controllers. This way we keep a low system utilization. Notice that γ is not a boolean parameter, but it can assume any value in $[0, 1]$.

Algorithm 3 Using γ to select the system behavior and solving contention

$SUM\_CS_{RATE} = \sum_i CS_{RATE}[i]$
if $SUM\_CS_{RATE} \leqslant CS_{MAX}$ **then**
  $CS = \gamma * CS_{ALL} + (1 - \gamma) * CS_{RATE}$
else
  $CS = CS_{ALL}$
end if

Since the value of CPU cores that the executor will acquire might not be the same as what the executor controller calculated, we need to update the state of the controller, to maintain consistency. If we do not, on the next control operation the controller would perform its computation based on a wrong previous state. This operation can be observed in Algorithm 4, where we update the value of $CSI_{OLD}[i]$ according to the value of $CS[i]$ that has been used to scale the executor's assigned number of CPU cores.

Algorithm 4 Updating CSI_{OLD} values for next iteration

for $i = 1$ **to** N **do**
  $CSI_{OLD}[i] = CS[i] - CSP[i]$
end for

The control operations are performed with a sampling time that is comparable to the one needed to the executor container to scale. In particular we know that a container scales in less than one second and thus using a sample time of one second is enough. It is important to notice that the `DistributeCores(.)` procedure mentioned in Algorithm 2 should be simple enough to be executed before the next control step, otherwise its delay would make the control useless.

3.6 RESOLVING RESOURCE CONTENTION

Contention resolution can be performed with different strategies, that can take into account static or dynamic characteristics of the applications, such as nominal rate or deadline. We identified some different strategies that solve contention exploiting those characteristics.

Allocating resources to Spark's executors of different applications can be seen as the preemptive online scheduling of sporadic tasks with arbitrary deadlines in a real time multiprocessor system. This way, scheduling tasks for execution has the effect of suggesting an allocation of CPU cores for the executors. We can take inspiration from real time algorithms to determine a possible partitioning of system resources amongst the executors. [16, 7, 9] proved that no optimal online algorithm for sporadic task sets with constrained or arbitrary deadlines can exists, since such an algorithm would require clairvoyance.

One of the most popular dynamic-priority planning based online algorithm is EDF scheduling [34], which uses deadlines to determine tasks priority. We can use xSpark applications' deadlines to determine a priority in allocating resources between the applications.

3.6.1 *Earliest Deadline First "All"*

With this strategy we allocate all the available resources to the application whose deadline is the closest, following an Earliest Deadline First (EDF) approach, even though the controller might have asked for less resources than the system maximum. In general, we want complete the execution of an application before allocating resources to the next one, and we want to complete the applications in a certain order, defined by their deadlines. It is possible that only one application is running at a given time.

Algorithm 5 shows how the allocation is performed. As we can see, the inputs to the algorithm are the maximum allocable cores of the system (max_cores) and, for each application, the time to complete (ttc_i) and the remaining tasks (rt_i).

An example of allocation is shown in Table 3.2, where we have contention between three different applications with different requests for cores, and different times to complete the execution. The system's total number of cores is 16. In this case we suppose that all the applications have enough tasks and can execute using all the system cores. We can see that only the application whose available time to complete is the smallest has acquired all the resources available in the machine.

APP	rc_i	ttc_i	cc_i
A	10.00	50	16.00
B	8.00	60	0.00
C	12.00	70	0.00

Table 3.2: Example of allocation using EDF "All". From left to right: the application (*App*), requested cores (rc_i), time available to complete (ttc_i), corrected cores assigned (cc_i).

Algorithm 5 Earliest Deadline First "All" core allocation

$apps \leftarrow [app_0, app_1, ...]$
$ttc \leftarrow [ttc_0, ttc_1, ...]$
$remaining\_tasks[rt_0, rt_1, ...]$
$corrected\_c \leftarrow [0, 0, ...]$
$remaining\_c \leftarrow max\_cores$
while $remaining\_c > 0$ **do**
  $app \leftarrow argmin_{app}^{apps}(ttc[app])$
  $assigned\_c \leftarrow min(remaining\_c, remaining\_tasks[app])$
  $remaining\_c \leftarrow remaining\_c - assigned\_c$
  $corrected\_c[app] \leftarrow assigned\_c$
  $apps \leftarrow apps \setminus app$
end while
return $corrected\_c$

3.6.2 *Earliest Deadline First "Pure"*

This strategy allocates resources to applications following a priority that is associated to the applications. The priority of an application is given by its remaining time to complete: the shorter the time is, the higher the priority. We allocate resources to the sorted applications according to their priority, reserving their desired number of CPU cores if available. With this strategy it is possible that some applications be paused since they have low priority.

Algorithm 6 shows how the allocation is performed. Inputs to the algorithm are the maximum allocable cores of the system (max_cores) and, for each application, the time to complete (ttc_i) and the cores (rc_i) requested by the executor's controller.

An example allocation is shown in Table 3.3, where three different applications with increasing remaining time to complete are in a state of contention. As a result, the application with the shortest time to complete will acquire all the cores it asked for, then the second one will allocate all the remaining cores of the system, because they are less than the requested ones, and the last application will see no cores granted for its execution.

APP	rc_i	ttc_i	cc_i
A	10.00	50	10.00
B	8.00	60	6.00
C	12.00	70	0.00

Table 3.3: Example of allocation using EDF "Pure". From left to right: the application (*App*), requested cores (rc_i), time available to complete (ttc_i), corrected cores assigned (cc_i).

Algorithm 6 Earliest Deadline First "Pure" core allocation

```
apps ← [app_0, app_1, ...]
ttc ← [ttc_0, ttc_1, ...]
requested_c ← [rc_0, rc_1, ...]
corrected_c ← [0, 0, ...]
remaining_c ← max_cores
while apps is not empty do
    app ← argmin_app^apps(ttc[app])
    assigned_c ← min(remaining_c, requested_c[app])
    remaining_c ← remaining_c − assigned_c
    corrected_c[app] ← assigned_c
    apps ← apps \ app
end while
return corrected_c
```

3.6.3 *Earliest Deadline First "Proportional"*

In this case, we assign each application a weight that is related to its remaining time to complete ttc_i. The weight is calculated as

$$w_i = 1 - \frac{ttc_i - min\_ttc + 1}{\sum_j^I [ttc_j - min\_ttc + 1]}$$

where $min\_ttc = min_i(ttc_i)$ and $i \in I$, the set of applications running at the given time. If there is not a huge difference in terms of remaining time to complete, all the applications will acquire a portion of the available resources.

Application weights are used in Algorithm 7 to assign cores proportionally to the weights of the applications (w_i) taking into account the requested number of cores (rc_i) and the system total allocable cores (max_cores). It is important to provide the applications' requested cores because we do not want to allocate more resources than the useful ones to applications that do not need them, even though their calculated weight is higher than the one of other applications. In this way, when all the applications are requesting to obtain the entire set of CPU cores, the resulting allocation is directly proportional to their weight. On the other hand, if one application is requesting only one

APP	rc_i	ttc_i	w_i	cc_i
A	10.00	50	0.97	7.75
B	8.00	60	0.67	5.35
C	12.00	70	0.36	2.90

Table 3.4: Example of allocation using EDF "Proportional". From left to right: the application (*App*), requested cores (rc_i), time available to complete (ttc_i), weight (w_i), corrected cores assigned (cc_i).

core but its weight is high in a way that, for example, it could allocate up to half the available cores, the application will still receive only one core, because it does not need more to proceed at the desired rate. We assume that the controller will not lie about the needing of a certain amount of CPU cores.

An example of allocation is shown in Table 3.4. As we can see, the initial situation described is the same as in the previous examples in Table 3.2 and 3.3, with three applications in a contention state running in a system with 16 allocable cores. The main difference with respect to the previous cases is that thanks to the allocation proportional to a weight, now all the three applications will see a certain amount of cores allocated to them, none of them will be paused in order to give precedence to the previous ones.

3.6.4 *Proportional*

This is the most basic way to allocate the available cores, after collecting the requests of all the applications in terms of desired cores (rc_i), the weight is simply calculated as

$$w_i = \frac{rc_i}{\sum_j^I rc_j}$$

where $i \in I$, the set of applications running at the given time. With this strategy, we obtain a fair distribution of the resources according to the requests of the applications, we do not try to improve the performances of a particular one with respect to another.

Application weights are used in Algorithm 7 to assign cores proportionally to the weights, as described in Section 3.6.3.

An example of allocation is shown in Table 3.5, where we have three applications in a contention state in a system with 16 allocable cores. As we can see, the weight is directly proportional to the amount of cores requested the applications, resulting in a final corrected allocation that is also proportional to the amount of requested cores.

APP	rc_i	w_i	cc_i
A	10.00	0.33	8.00
B	8.00	0.27	6.40
C	12.00	0.40	9.60

Table 3.5: Example of allocation using Proportional. From left to right: the application (*App*), requested cores (rc_i), weight (w_i), corrected cores assigned (cc_i).

3.6.5 *Speed*

This strategy takes into account the speed of the application measured in terms of average nominal rate. Nominal rate is the number of input records processed per second per core, this value is obtained from the profiling phase and is calculated per stage. In order to compare two applications, we decided to calculate an application average nominal rate as

$$anr_i = \frac{\sum_s^S nominal\_rate_s \cdot w_s}{\sum_s^S w_s}$$

where $s \in S$, the set of stages of the application, and w_s is the weight of the stage, also coming from the profiling phase. Knowing the average nominal rate of the application, we can use it to calculate the weight of the running application as

$$w_i = \frac{ANR}{anr_i}$$

where $i \in I$, the set of applications running at the given time, and ANR is the average nominal rate constant calculated taking into account the nominal rate of different kind of Spark applications.

Application weights are used in Algorithm 7 to assign cores proportionally to the weights, as explained in Section 3.6.3.

An example of allocation is shown in Table 3.6, where we have three different applications, whose total amount of requested cores (30) that is larger than the system total allocable cores (16). The ANR has been set to 1.000.000. Since applications A and B have the same nominal rate, they will have the same weight. Instead, C has a lower nominal rate, in particular it is half the nominal rate of A and B, and we see that its weight is the double with respect to the one of A and B. As a result, C will obtain half of the resources of the system, meanwhile A and B one quarter each.

Algorithm 7 Allocate cores to applications given their weights and desired cores

```
apps ← [app_0, app_1, ...]
requested_c ← [rc_0, rc_1, ...]
corrected_c ← [0, 0, ...]
weights ← [w_0, w_1, ...]
remaining_c ← max_cores
while apps is not empty and remaining_c > 0 do
  completed_apps ← [ ]
  total_assigned_c ← 0
  total_weight ← ∑_a^apps weights[a]
  for all app ∈ apps do
    assignable_c ← (weights[app] / total_weight) * remaining_c
    assigned_c ← min(assignable_c, requested_c[app])
    corrected_c[app] ← corrected_c[app] + assigned_c
    if assignable_c ⩾ requested_c[app] then
      completed_apps ← completed_apps + app
    end if
    requested_c[app] ← requested_c[app] − assigned_c
    total_assigned_c ← total_assigned_c + assigned_c
  end for
  remaining_c ← remaining_c − total_assigned_c
  apps ← apps \ completed_apps
end while
return corrected_c
```

APP	rc_i	anr_i	w_i	cc_i
A	10.00	1.000.000	1.00	4.00
B	8.00	1.000.000	1.00	4.00
C	12.00	500.000	2.00	8.00

Table 3.6: Example of allocation using Speed. From left to right: the application (*App*), requested cores (rc_i), nominal rate (anr_i), weight (w_i), corrected cores assigned (cc_i).

4 IMPLEMENTATION

In this chapter we show the implementation details of the new components that have been added to xSpark and explain which modifications have been done to already existing ones.

4.1 HEURISTIC

The heuristic to be used in xSpark is determined by the value configuration parameter `spark.control.heuristic` and is an implementation of the class `HeuristicBase`, whose class diagram is shown in Figure 4.1. As we can see in Listing 4.1, in xSpark the default heuristic that is used is `HeuristicControl`, but other heuristics have been implemented: `HeuristicFixed` and `HeuristicUnlimited`.

Listing 4.1: `ControlEventListener`: loading the correct heuristic implementation.

```
val heuristicType = conf.getInt("spark.control.heuristic", 0)
val heuristic: HeuristicBase =
if (heuristicType == 1 && conf.contains("spark.control.stagecores
    ") && conf.contains("spark.control.stagedeadlines") && conf.
    contains("spark.control.stage"))
        new HeuristicFixed(conf)
else if (heuristicType == 2)
        new HeuristicControlUnlimited(conf)
else
        new HeuristicControl(conf)
```

`HeuristicFixed`, which has been used for testing purposes, allows the user to specify through Spark configuration parameters a fixed core allocation and stage duration for each of the stage of the running application. The user needs to specify three different lists, that must have the same length, representing the stages (`spark.control.stage`), their allocated cores (`spark.control.stagecores`) and their deadline (`spark.control.stagedeadlines`). These lists are transformed into maps in order to have a faster access to their value, as shown in Listing 4.2. When the heuristic methods are called, instead of calculating the allocations of cores and the deadlines of the stages as in the base version, this implementation simply reads the values from the previously created maps.

Listing 4.2: `HeuristicFixed`: using values provided by the user.

```
\\ parse configuration values
```

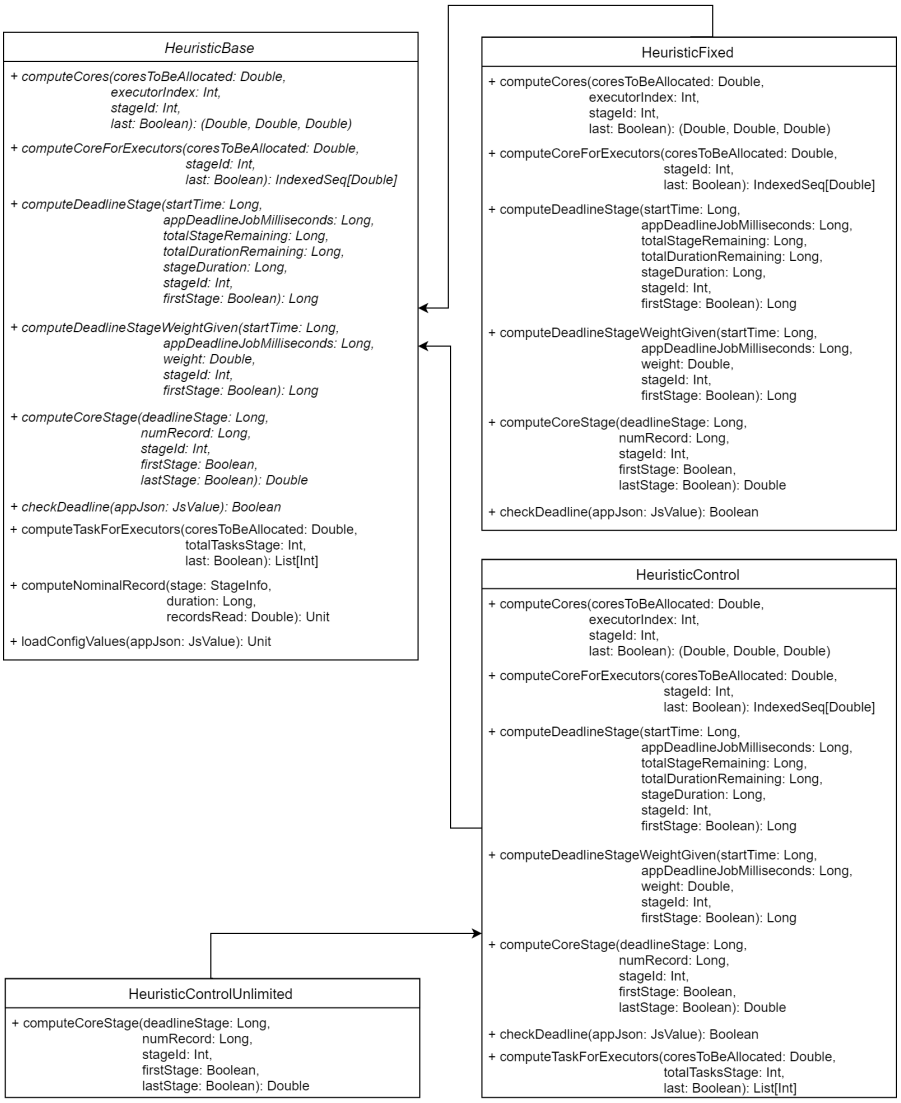


Figure 4.1: Class diagram of Heuristic related classes.

```
val stagesToFix: List[Int] = conf.get("spark.control.stage").
    replace("[", "").replace("]","").split(',').toList.map(_.trim
    ).map(_.toInt)
val stageCores: List[Double] = conf.get("spark.control.stagecores
    ").replace("[", "").replace("]", "").split(',').toList.map(_.
    trim).map(_.toDouble)
val stageDeadlines: List[Long] = conf.get("spark.control.
    stagedeadlines").replace("[", "").replace("]", "").split(',')
    .toList.map(_.trim).map(_.toLong)

\\ create maps
val stageToCoresConf = (stagesToFix zip stageCores).toMap
val stageToDeadlinesConf = (stagesToFix zip stageDeadlines).toMap
```

`HeuristicUnlimited` instead simply extends `HeuristicControl` and reimplements the method `computeCoreStage`. The goal is to avoid requesting the allocation of a number of CPU cores that is larger than the ones available in the system (Listing 4.3). This is done by comparing the result of the same method from the parent class with the available resources, calculated using the spark configuration parameters `spark.control.maxexecutors` and `spark.control.coreforvm`, that provide respectively the maximum number of executors per application and the number of cores each of them can allocate.

Listing 4.3: `HeuristicUnlimited`: adjusting the number of stage cores.

```
override def computeCoreStage(deadlineStage: Long = 0L, numRecord
    : Long = 0L, stageId: Int = 0, firstStage : Boolean = false,
    lastStage: Boolean = false): Double = {
        val requestedCores = super.computeCoreStage(deadlineStage
            , numRecord, stageId, firstStage, lastStage)
        if (requestedCores > coreForVM * numMaxExecutor){
                coreForVM * numMaxExecutor
        } else {
                requestedCores
        }
}
```

4.2 LAUNCHING APPLICATIONS

As introduced in Section 3.3, we needed to modify how xSpark launches executors on its workers. In the original implementation, the maximum number of usable cores by an executor is the number of cores free that its worker has. This obviously is not what we are interested in, since we want to have a soft boundary between the cores allocated to each of the executors of a worker. For example, in the standard implementation, the first executor launching is potentially allocating all the cores available in the machine, in this way, the next application will have to wait until the previous one finishes, in order to allocate

cores for its executor. We want to avoid this phenomena, since we are interested in giving the controller the job of controlling the allocation of the CPU cores.

The simplest solution is to keep track of the amount of cores assigned to a particular application on a worker, as seen in Listing 4.4. This values are updated every time an executor is added, removed or scaled.

Listing 4.4: `WorkerInfo`: keeping track of cores assigned to applications.

```
// cores used by an application
val applicationIdToCoresUsed = new mutable.HashMap[String, Int]()
    .withDefaultValue(0)

// old implementation
def coresFree: Int = cores - coresUsed

// new implementation
def coresFree(applicationId: String): Int = cores -
    applicationIdToCoresUsed(applicationId)
```

Notice that the cores used by an application are initialized at a zero value, this because we want an application to be able to allocate all the cores available on the machine, without taking into account if other applications have already launched their executors. As a consequence, we use the newly defined `coresFree(applicationId)` when starting new executors for an application, in this way cores for executors can be allocated as if there are no other applications running. As we can see in Listing 4.5, when starting an executor we compare the number of cores that the executor should have with the amount of cores already assigned to that application on the selected worker.

Since the same soft boundary is not applicable on the memory point of view, because we cannot resize the memory allocated to a JVM process without restarting it, the free memory requested for an executor is compared with the effectively free memory that the worker can offer.

Listing 4.5: `Master`: launching executors on workers.

```
private def startExecutorsOnWorkers(): Unit = {
// Right now this is a very simple FIFO scheduler. We keep trying
     to fit in the first app in the queue, then the second app,
    etc.
for (app <- waitingApps if app.coresLeft > 0) {
        val coresPerExecutor: Option[Int] = app.desc.
            coresPerExecutor
        // Filter out workers that don't have enough resources to
             launch an executor
        val usableWorkers = workers.toArray.filter(_.state ==
            WorkerState.ALIVE).filter(worker => worker.memoryFree
             >= app.desc.memoryPerExecutorMB && worker.coresFree(
```

```
            app.id) >= coresPerExecutor.getOrElse(1)).sortBy(_.
            coresFree).reverse
        val assignedCores = scheduleExecutorsOnWorkers(app,
            usableWorkers, spreadOutApps)

        // Now that we've decided how many cores to allocate on
            each worker, let's allocate them
        for (pos <- 0 until usableWorkers.length if assignedCores
            (pos) > 0) {
                allocateWorkerResourceToExecutors(app,
                    assignedCores(pos), coresPerExecutor,
                    usableWorkers(pos))
        }
}
}
```

As we can see, applications are scheduled in a First In First Out (FIFO) order. This can be reimplemented in a simple way in order to take care of a priority value of the applications or to take into account their deadline, giving precedence to applications with a shorter deadline.

4.3 SCALABLE OFF-HEAP MEMORY

In order to support the scaling of off-heap memory size, we needed to modify xSpark `MemoryManager`. Spark provides two implementations of it, `StaticMemoryManager` and `UnifiedMemoryManager` (Figure 4.2). We extended the base abstract class in order to include a method that will allow us to resize the off-heap memory (Listing 4.6).

Listing 4.6: `MemoryManager`: resizing off-heap memory.

```
def resizeOffHeapMemory(newSize: Long): Unit = {
        return
}
```

Since `StaticMemoryManager` does not support off-heap allocation, there was no need to implement the method in that class.

Instead, in order to use the functionality in `UnifiedMemoryManager`, that is the default Memory Manager in Spark since version 1.6.0, we simply needed to use the already existing methods used to shrink memory and storage pools, paying attention to obtain the desired result (Listing 4.7).

Listing 4.7: `UnifiedMemoryManager`: implementation of resizeOffHeapMemory.

```
/**
* Resizes the offheap pools so that the total memory is updated
    to the given value
* @param newSize total memory of offheap pools
```

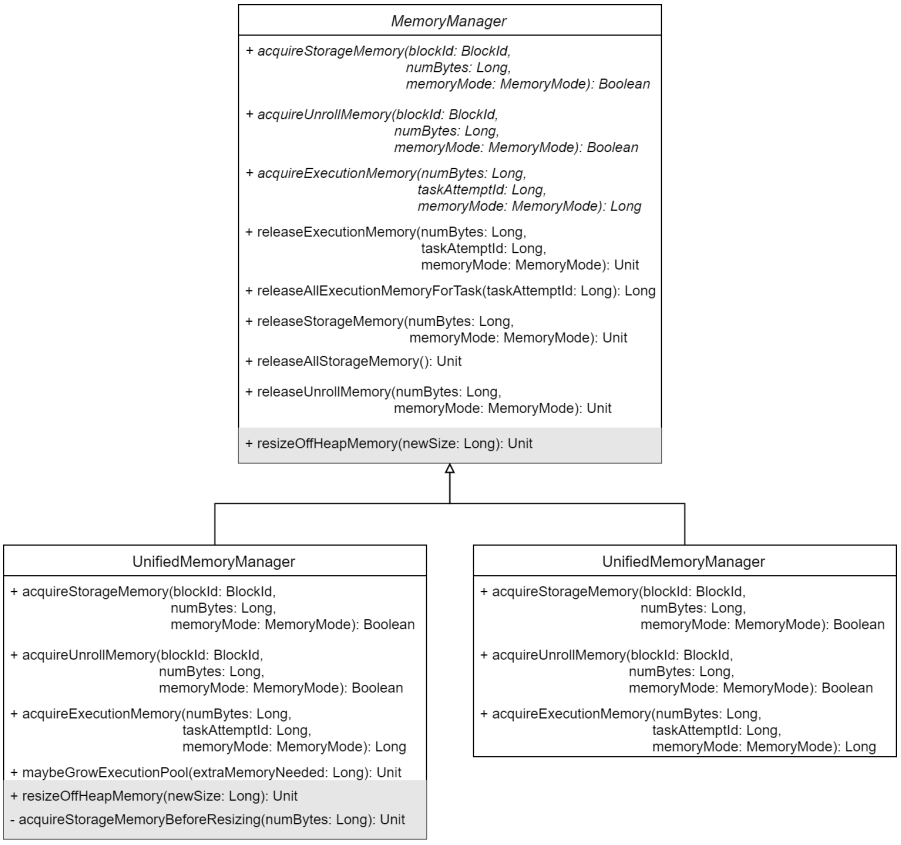


Figure 4.2: Class diagram of MemoryManager related classes. Methods with a grey background are those that have been introduced in order to perform dynamic resizing of off-heap memory.

```
*/
override def resizeOffHeapMemory(newSize: Long): Unit = {
        logInfo("Resizing offheap memory to "+newSize+" from "+
            currentMaxOffHeapMemory)
        if (maxOffHeapMemory == 0) {
                // nothing to do if we are not using off heap
                    memory :)
                return
        }
        val delta = currentMaxOffHeapMemory - newSize
        if (delta > 0) {
                // try to prevent disk swapping by resizing pools
                acquireStorageMemoryBeforeResizing(delta)
                // do the swapping if necessary
                offHeapStorageMemoryPool.freeSpaceToShrinkPool(
                    delta)
                // decrement pool size
                offHeapStorageMemoryPool.decrementPoolSize(delta)
                currentMaxOffHeapMemory = newSize
        } else if (delta < 0) {
                // increment pool size
                offHeapStorageMemoryPool.incrementPoolSize(math.
                    abs(delta))
                currentMaxOffHeapMemory = newSize
        }
}

/**
* Borrows memory from the execution pool for the storage pool
* @param numBytes memory to borrow
*/
private def acquireStorageMemoryBeforeResizing(numBytes: Long) =
    synchronized {
        if (numBytes > offHeapStorageMemoryPool.memoryFree) {
                val memoryBorrowedFromExecution = Math.min(
                    offHeapExecutionMemoryPool.memoryFree,
                    numBytes)
                offHeapExecutionMemoryPool.decrementPoolSize(
                    memoryBorrowedFromExecution)
                offHeapStorageMemoryPool.incrementPoolSize(
                    memoryBorrowedFromExecution)
        }
}
```

In case of decreasing size, `acquireStorageMemoryBeforeResizing` tries to acquire as much memory as possible from the off-heap Execution Memory Pool for the Storage Memory Pool. After this operation there are two possible situation, depending on how much free memory the Storage Memory Pool has acquired. If the free off-heap memory of the Storage pool is larger than `delta`, when calling the method `freeSpaceToShrinkPool` the result is that the Storage pool decreases without touching the blocks already in memory. Instead, if the free

memory is smaller, calling the method will result in dropping blocks from memory in order to free enough space to perform the shrinking. These blocks are either discarded or stored to disk, according to their `StorageLevel`. Once enough free memory in the Storage pool is available, the pool size is reduced.

Instead, if the size is growing, we can simply extend the Storage pool, because thank to the implementation of `UnifiedMemoryManager`, we do not have hard boundaries between the two pools and Execution pool can acquire memory from the other one when needed.

Resizing is performed when an executor belonging to a Worker is starting or terminating. This events are initiated with the two messages `LaunchExecutor` and `KillExecutor` received by the Worker and sent by the Master. When a new executor is launching, we calculate the new distribution of off-heap memory taking into account the portion of free memory that will be allocated to the JVM of the starting executor, as seen in Listing 4.8. Instead, when the executor is stopping, we take into account the memory assigned to the heap of the executor that is being shut down, shown in Listing 4.9.

Since the executors are running in a different JVM inside a Docker container, we need to define another message in order to be able to ask them to resize their off-heap allocation. The message is `ResizeOffHeapMemory` and has one parameter that corresponds to the off-heap memory size that needs to be set. The `Worker` will send the messages to each executor's proxy which will then deliver it to the `CoarseGrainedExecutorBackend` of the recipient executor.

Listing 4.8: `Worker`: resizing memory when launching an executor.

```
override def receive: PartialFunction[Any, Unit] = synchronized {
...
case LaunchExecutor(masterUrl, appId, execId, appDesc, cores_,
    memory_) =>
        ...
        logInfo("Asked to launch executor %s/%d for %s".format(
            appId, execId, appDesc.name))
        val offHeapMemory: Long = (memoryFree - memory_) / (
            execIdToProxy.size + 1)
        // off-heap memory in mega bytes
        execIdToProxy.foreach { case (id, proxy) =>
                proxy.proxyEndpoint.send(ResizeOffHeapMemory(
                    offHeapMemory*1000000))
        }
        ...
}
```

Listing 4.9: `Worker`: resizing memory when killing an executor.

```
override def receive: PartialFunction[Any, Unit] = synchronized {
...
```

```
case KillExecutor(masterUrl, appId, execId) =>
        ...
        logInfo("Asked to kill executor " + fullId)
        ...
        if (execIdToProxy.size > 0) {
                val offHeapMemory: Long = (memoryFree.toLong +
                    executor.memory) / execIdToProxy.size
                // off-heap memory in mega bytes
                execIdToProxy.foreach { case (id, proxy) =>
                        proxy.proxyEndpoint.send(
                            ResizeOffHeapMemory(offHeapMemory
                            *1000000))
        }
        ...
}
```

Notice that, when sending the message after an executor is starting, we are not sending the message to the new executor, indeed it has not yet started and thus it is not able to receive the message. In order to set the correct off-heap memory for this one, we exploited the fact that all the communication between Executor and Master is intercepted by the `ControllerProxy` of the executor. In this way, when the Driver replies to the Executor's message `RetrieveSparkProps`, that is used to send a copy of Spark properties from one to another, we can inject the correct value for the off-heap memory of the starting executor, shown in Listing 4.10. `offheapBytes` value is set inside `ControllerProxy` when it is instantiated, using the same value that has been calculated before in order to scale other executors.

Listing 4.10: `ControllerProxy`: updating the value of off-heap memory.

```
override def receiveAndReply(context: RpcCallContext):
    PartialFunction[Any, Unit] = {
        ...
        case RetrieveSparkProps =>
                val sparkProperties = driver.get.askWithRetry[Seq
                    [(String, String)]](RetrieveSparkProps)
                val sparkPropertiesUpdated = sparkProperties.map{
                    case (prop, value) =>
                        if (prop == "spark.memory.offHeap.size"){
                                (prop, offheapBytes.toString)
                        } else {
                                (prop, value)
                        }
                }
        context.reply(sparkPropertiesUpdated)
```

With this implementation, we obtain that at a given time, the total memory allocated for heap and off-heap is not greater than the system one.

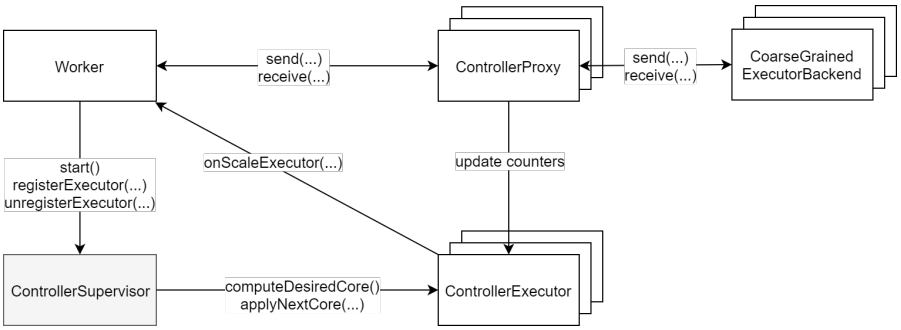


Figure 4.3: The interactions between the old components and the newly introduced `ControllerSupervisor`.

4.4 CONTROLLER

In order to implement the new controller proposed in Section 3.5, we needed to modify the previous implementation of xSpark `ControllerExecutor`, because we needed to synchronize the scaling of the different executors that are running on a machine in order to keep the total number of allocated cores under the system maximum. To do so, we implemented a supervisor of the different executors called `ControllerSupervisor`, whose goal is to retrieve the next allocation of cores from the different executors' controller, if necessary find a different feasible allocation and in the end apply the new core values. In Figure 4.3 is shown the new structure, showing the presence of multiple executors' controller and proxy that interact with a single instance of `Worker` and `ControllerSupervisor`.

`ControllerExecutor` class now presents two different methods that are used by `ControllerSupervisor`. In order to obtain the number of cores that could make the executor follow the desired progress rate, the supervisor needs to use the method `computeDesiredCore()` provided by each executor controller, as displayed in Listing 4.11. First of all, the set point value (SP) is updated according to the *deadline* of the current stage and the sampling time (Ts). If $SP \geqslant 1$ it means that the deadline has already elapsed and the computation should already be finished, this is why we request to allocate all the cores. Instead, if $SP < 1$, we use the PI controller implemented in the method `nextAllocation()` in order to obtain a value of CPU cores that should be applied in order to keep up with the desired progress rate.

Listing 4.11: `ControllerExecutor`: finding the next core allocation.

```
def nextAllocation(): Double = {
        csp = K * (SP - (completedTasks / tasks))
        val csi = csiOld + K * (Ts.toDouble / Ti) * (SP - (
            completedTasks / tasks))
        val cs = math.min(math.max(coreMin.toDouble, csp + csi),
            (tasks-completedTasks))
        cs
```

```
}

def computeDesiredCore(): Double = {
        if (SP < 1.0) SP += Ts.toDouble / deadline.toDouble
        var nextCore: Double = coreMin
        if (SP >= 1.0) {
                SP = 1.0
                nextCore = coreMax
        } else {
                nextCore = nextAllocation()
        }
        nextCore
}
```

The next method used by the supervisor and shown in Listing 4.12 is `applyNextCore(nextCore, requestedCore)`. The implementation allows to correct the value of the core to be applied (*nextCore*) in order a multiple of the granularity offered by Docker using the core quantum (CQ) constant, that is the smallest fraction of core that is assignable to a container. Once the new core is calculated, we update the value of *csiOld* that will be used by the next iteration of the PI controller. Last thing to do is to ask the worker to scale its executor, which might not be needed according to the value of previously assigned core (*oldCore*)

Listing 4.12: `ControllerExecutor`: applying next core and updating old values.

```
def applyNextCore(nextCore: Double, requestedCore: Double) = {
        // match core quantum
        val cs = math.ceil(nextCore / CQ) * CQ
        // store old value
        csiOld = cs - csp
        // scale executor
        if (cs != oldCore) {
                oldCore = cs
                worker.onScaleExecutor(applicationId, executorId,
                     cs)
        }
}
```

As we can see, execution of the methods of the controller executor are no longer scheduled independently, instead `ControllerSupervisor` is in charge of scheduling the execution of the control loop every sampling time, defined by the constant Ts. This supervisor is instantiated at worker creation time and has been implemented in different flavors that have been introduced in Sections 3.6.1 to 3.6.5. As we can see in Listing 4.13, the value of the configuration parameters `spark.control.supervisor.gamma` and `spark.control.supervisor` determine respectively the value of the γ parameter and the implementation of `ControllerSupervisor` to be used. We decided that the de-

fault combination is using γ equal to one and the proportional version of the supervisor.

Listing 4.13: `Worker`: initialization of `ControllerSupervisor`.

```
val supervisorGamma = conf.getDouble("spark.control.supervisor.
    gamma", 1d)
val supervisorType = conf.get("spark.control.supervisor", "
    proportional").toLowerCase
val supervisor: ControllerSupervisorAbstract = supervisorType
    match {
        case "edf_pure" => new ControllerSupervisorPureEDF(cores,
             Ts, supervisorGamma)
        case "edf_proportional" => new
            ControllerSupervisorProportionalEDF(cores, Ts,
            supervisorGamma)
        case "edf_all" => new ControllerSupervisorEDFAll(cores,
            Ts, supervisorGamma)
        case "speed" => new ControllerSupervisorSpeed(cores, Ts,
            supervisorGamma, conf.getLong("spark.control.
            avgnominalrate", 1000000))
        case "mixed_speed_edf" => new
            ControllerSupervisorMixedSpeedEDF(cores, Ts,
            supervisorGamma, conf.getLong("spark.control.
            avgnominalrate", 1000000))
        // default is proportional
        case _ => new ControllerSupervisorProportional(cores, Ts,
             supervisorGamma)
}
```

`ControllerSupervisor` keeps track of the currently active executors running on the same worker, using a hash map. The worker will register and unregister executors with the supervisor using two different methods that are shown in Listing 4.14. Notice that both methods are synchronized on the hash map, because we want to avoid the possibility of concurrent edits to the data structure.

Listing 4.14: `ControllerSupervisor`: keeping track of active executors.

```
def registerExecutor(applicationId: ApplicationId, executorId:
    ExecutorId, controllerExecutor: ControllerExecutor) = {
activeExecutors.synchronized {
        activeExecutors += (((applicationId, executorId),
            controllerExecutor))
        logInfo("Registering new executor " + applicationId + "/"
             + executorId + ", total executors " +
            activeExecutors.size)
}
}

def unregisterExecutor(applicationId: ApplicationId, executorId:
    ExecutorId) = {
activeExecutors.synchronized {
```

```
        activeExecutors -= ((applicationId, executorId))
        logInfo("Unregistering executor " + applicationId + "/" +
             executorId + ", total executors " + activeExecutors.
            size)
}
}
```

The worker will inform the supervisor about the change in the set of running executor according to the messages it receives from the master. When it receives `InitControllerExecutor`, which is sent every time a stage is assigned to an executor, first of all a new `ControllerExecutor` is instantiated and the supervisor is notified; instead, when the message is `KillExecutor`, which in general signals the ending of the application, the controller is removed from the supervisor and the it is terminated.

`ControllerSupervisor` main loop performs a series of operations in order to determine the allocation of CPU cores for each of the running application. As we can see in Listing **??**, first of all we ask each of the executors' controller their requested allocation in order to follow the desired progress rate (*csForRate*). Next, we calculate an allocation of cores that distributes all the cores available in the system (*csAllCores*), in this way we want to have a complete utilization of the machine CPU resources. This operation is performed using the method `correctCores(cores)` that is implemented in different way in the various subclasses of `ControllerSupervisor`. At this point, we check if there is contention, remembering that we are in a contention state when the sum of the cores requested by all the applications (*sumCoresForRate*) is greater than the ones the machine can offer (*maximumCores*). In case of contention, we apply the cores that have been defined in *csAllCores*, otherwise we use the parameter γ to determine an allocation that can span from *csAllCores* to *csForRate*, according to the value of the previous parameter.

Listing 4.15: `ControllerSupervisor`: main loop

```
// obtain cores to follow the progress rate
activeExecutors.foreach { case (id, controllerExecutor) =>
        var desiredCore = controllerExecutor.computeDesiredCore()
        csForRate += ((id, desiredCore))
}

val sumCoresForRate = csForRate.values.sum

// calculate an allocation that distributes all the cores
if (sumCoresForRate == 0){
        csAllCores = csForRate.map{case (id,_) => (id, 0d)}
} else {
        var correctedCores = correctCores(csForRate)

        val correctedCoresSum = correctedCores.values.sum
```

```
        if((0.99 * maximumCores) > correctedCoresSum &&
            correctedCoresSum > 0){
                correctedCores = correctedCores.map{case (id,core
                    ) => (id, (core/correctedCoresSum)*
                    maximumCores)}
        }
csAllCores = correctedCores
}

// check if there is contention
if (sumCoresForRate <= maximumCores){
        // no contention, use gamma
        cs = csForRate.map{case (id, thisCsForRate) => (id, gamma
            *csAllCores.get(id).get + (1-gamma)*thisCsForRate)}
} else {
        // contention
        cs = csAllCores
}

// apply cores
activeExecutors.foreach { case (id, controllerExecutor) =>
        controllerExecutor.applyNextCore(cs(id), csForRate(id))
}
```

`ControllerSupervisor` main loop is started when the Worker has been fully initialized, before accepting applications. All the `ControllerSupervisor` subclasses implement a different version of the method `correctCores(cores)` according to the various strategies proposed in Section 3.5. The reimplemented classes are shown in the class diagram in Figure 4.4. Here we discuss their implementation details.

4.4.1 *ControllerSupervisorEDFAll*

As we have introduced in Section 3.6.1, the goal of this implementation is to allocate as much resources as it can handle to the application who is supposed to end first among the set of all executing applications. According to the code shown in Listing 4.16, this is achieved by obtaining the *remainingTimeToComplete* of the various applications, as the difference between *currentTimestamp* and the timestamp corresponding to their deadline, provided by the `ControllerExecutor` of the application (*controllerExecutor.deadlineAppTimestamp*). Then we determine the number of usable cores by an application, since in xSpark we can allocate only up to one core to a task, this number is determined by the amount of remaining tasks for an application, calculated as the difference between the assigned tasks and the already completed ones, both values provided by the controller (*controllerExecutor.tasks - controllerExecutor.completedTasks*). We determine the value of *maxUsableCores* as the minimum between the system cores (*maximumCores*) and the previously calculated remaining number of tasks.

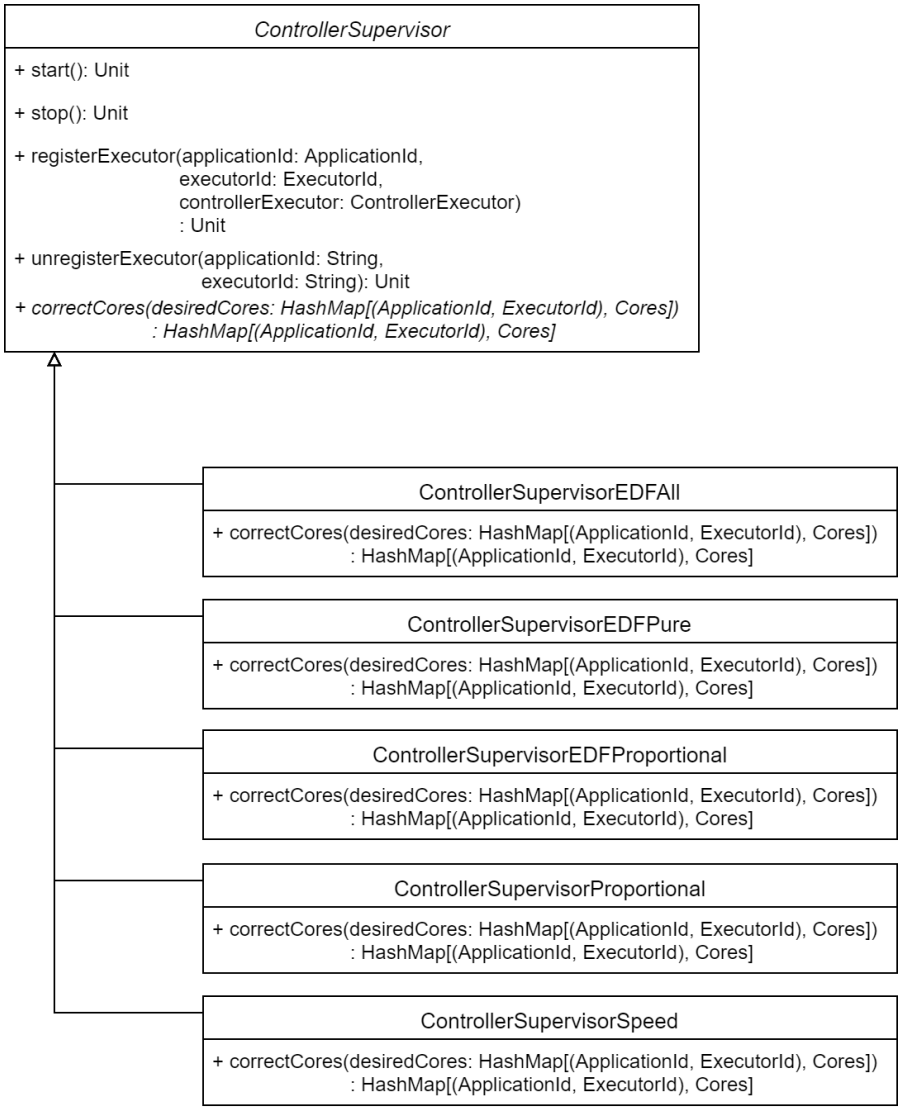


Figure 4.4: Class diagram of ControllerSupervisor and its subclasses.

In order to allocate cores, we sort the applications according to their remaining time to complete execution and begin allocating their usable cores. Notice that if the first application allocates all the available cores, all the others will simply result in having zero cores allocated to their execution. This does not happen only when the first application to be analyzed can allocate only an amount of CPU cores that is smaller than the system available cores.

Listing 4.16: `ControllerSupervisorEDFAll`: implementation of method correctCores.

```
var correctedCores = new mutable.HashMap[(ApplicationId,
    ExecutorId), Cores]()
val currentTimestamp = System.currentTimeMillis()
var remainingCores = maximumCores.toDouble
val remainingTimeToComplete = activeExecutors.map { case (id,
    controllerExecutor) =>
(id, controllerExecutor.deadlineAppTimestamp - currentTimestamp)
}
val maxUsableCores = activeExecutors.map { case (id,
    controllerExecutor) =>
(id, math.min(controllerExecutor.tasks - controllerExecutor.
    completedTasks, maximumCores))
}

val sortedKeys = remainingTimeToComplete.toSeq.sortBy(_._2)

sortedKeys.foreach({case (id, _) => {
if (maxUsableCores(id) <= remainingCores){
        correctedCores += ((id, maxUsableCores(id)))
        remainingCores -= maxUsableCores(id)
} else {
        correctedCores += ((id, remainingCores))
        remainingCores = 0
}
}})

return correctedCores
```

4.4.2 *ControllerSupervisorEDFPure*

The proposed implementation is similar to the one we have discussed for `ControllerSupervisorEDFAll`, as we can see in Listing 4.17. The main difference is that we limit the allocable cores of each applications to the number of cores that the executor's controller has requested, as introduced in Section 3.6.2. As we can see, we order the applications according to their *remainingTimeToComplete* and allocate to each of them their requested number of cores, passed as a parameter to the method in the HashMap *desiredCores*. In this way, the result-

ing allocation for each application is a value between zero and the value contained in *desiredCores*.

Listing 4.17: `ControllerSupervisorEDFPure`: implementation of method correctCores.

```
var correctedCores = new mutable.HashMap[(ApplicationId,
    ExecutorId), Cores]()
val currentTimestamp = System.currentTimeMillis()
var remainingCores = maximumCores.toDouble
val remainingTimeToComplete = activeExecutors.map { case (id,
    controllerExecutor) =>
(id, controllerExecutor.deadlineAppTimestamp - currentTimestamp)
}

val sortedKeys = remainingTimeToComplete.toSeq.sortBy(_._2)

sortedKeys.foreach({case (id, _) => {
        if (desiredCores(id) <= remainingCores){
                correctedCores += ((id, desiredCores(id)))
                remainingCores -= desiredCores(id)
        } else {
                correctedCores += ((id, remainingCores))
                remainingCores = 0
        }
}})

return correctedCores
```

4.4.3 *ControllerSupervisorEDFProportional*

In Listing 4.18 is shown the implementation of the variant discussed in Section 3.6.3. We want to allocate cores in a way that is proportional to the remaining time to complete the execution. As previously explained, first of all we want to know the applications *remainingTimeToComplete*. We want to also know which is the minimum time among all the applications, stored in *minTimeToComplete*. In order to take into account applications that might be late, whose time to complete is negative, we move the previously calculated times. In this way, the application whose deadline is the closest will have *trasled(id)* = 1. Then we calculate the weight of each of the applications as *deadlineWeight*. Notice that, if there is a single application running, we need to fix its weight value, because otherwise it would be zero. Once we have all the application weights, we apply the implementation of the Algorithm 7 that has been previously discussed. At each iteration, we assign the *remainingCores* to be allocated, which have been initialized to the system maximum, every time we assign cores to an application, we update the value of *remainingCores* and *desiredCoresLocal*, which represents the number of cores the application is still

asking for. Once an application obtains all the requested resources, its key is removed from both *desiredCoresLocal* and *deadlineWeight*, in order to be able to split the remaining cores at the next iteration to the remaining applications.

Listing 4.18: `ControllerSupervisorEDFProporional`: implementation of method correctCores.

```
val trasled: mutable.HashMap[(ApplicationId, ExecutorId), Double]
     = remainingTimeToComplete.map { case (id, ttc) =>
        (id, ttc - minTimeToComplete + 1)
}
var trasledSum = trasled.values.sum
var deadlineWeight = trasled.map{ case(id, trttc) =>
        (id, 1 - (trttc/trasledSum))
}
// prevent single app to get weight zero
if(deadlineWeight.size == 1){
        deadlineWeight = deadlineWeight.map{ case(id, _) => (id,
            1d)}
}

var desiredCoresLocal = desiredCores.clone()
var remainingCores = maximumCores.toDouble
while (!desiredCoresLocal.isEmpty && remainingCores > 0) {
        var completedExecutors: mutable.MutableList[(
            ApplicationId, ExecutorId)] = mutable.MutableList[(
            ApplicationId, ExecutorId)]()
        val totalDeadlineWeight: Double = deadlineWeight.values.
            sum
        var totalAssignedCores: Double = 0

        desiredCoresLocal.foreach { case (id, cores) => {
                val assignableCores = (deadlineWeight(id) /
                    totalDeadlineWeight) * remainingCores
                val assignedCores = math.min(assignableCores,
                    cores)
                // assign cores
                correctedCores(id) = correctedCores.getOrElse(id,
                     0.toDouble) + assignedCores
                // reduce asked cores
                desiredCoresLocal(id) = desiredCoresLocal(id) -
                    assignedCores
                // remove executor if given all the requested
                    cores
                if (assignableCores >= cores) {
                        completedExecutors += id
                }
                // update accumulators
                totalAssignedCores += assignedCores
                }
        }
        // update available cores to assign
```

```
        remainingCores -= totalAssignedCores
        // update executors requesting other cores
        completedExecutors.foreach { id => {
                desiredCoresLocal -= id
                deadlineWeight -= id
        }}
}

return correctedCores
```

4.4.4 *ControllerSupervisorProportional*

The strategy proposed in Section 3.6.4 is the one whose implementation is the simplest, as we can notice from Listing 4.19. What we need to do is to solve a simple proportion for each of the executor's requests

$$\text{desiredCores}(\text{id}) : \text{totalRequestedCores} = x : \text{maximumCores}$$

where x is the value of *correctedCores(id)*, *totalRequestedCores* is the sum of the requests from the controllers that are stored in *desiredCores* and *maximumCores* is the total system CPU cores.

Listing 4.19: `ControllerSupervisorProportional`: implementation of method correctCores.

```
var correctedCores = new mutable.HashMap[(ApplicationId,
    ExecutorId), Cores]()
val totalRequestedCores = desiredCores.values.sum
desiredCores.foreach { case (id, cores) =>
        correctedCores += ((id, (maximumCores /
            totalRequestedCores) * cores))
}

return correctedCores
```

4.4.5 *ControllerSupervisorSpeed*

As explained in Section 3.6.5, the main difference between the strategy implemented here and the one in `ControllerSupervisorEDFProportional` is the weight associated to each of the applications. Here the weight is stored in the HashMap *speed* and is calculated as the ratio between *avgNominalRate*, that is read from Spark configuration property `spark.control.avgnominalrate`, and the *nominalRateApp* provided by the executor controller.

Listing 4.20: `ControllerSupervisorSpeed`: implementation of method correctCores.

```
var correctedCores = new mutable.HashMap[(ApplicationId,
    ExecutorId), Cores]()
val speed = activeExecutors.map { case (id, controllerExecutor) =
    >
        (id, avgNominalRate / controllerExecutor.nominalRateApp)
}

var desiredCoresLocal = desiredCores.clone()
var remainingCores = maximumCores.toDouble

while (!desiredCoresLocal.isEmpty && remainingCores > 0) {
        var completedExecutors: mutable.MutableList[(
            ApplicationId, ExecutorId)] = mutable.MutableList[(
            ApplicationId, ExecutorId)]()
        val totalSpeed = speed.values.sum
        var totalAssignedCores: Double = 0

        desiredCoresLocal.foreach { case (id, cores) => {
                val assignableCores = (speed(id) / totalSpeed) *
                    remainingCores
                val assignedCores = math.min(assignableCores,
                    cores)
                // assign cores
                correctedCores(id) = correctedCores.getOrElse(id,
                     0.toDouble) + assignedCores
                // reduce asked cores
                desiredCoresLocal(id) = desiredCoresLocal(id) -
                    assignedCores
                // remove executor if given all the requested
                    cores
                if (assignedCores >= cores) {
                        completedExecutors += id
                }
                // update accumulators
                totalAssignedCores += assignedCores
                }
        }
        // update available cores to assign
        remainingCores -= totalAssignedCores
        // update executors requesting other cores
        completedExecutors.foreach { id => {
        desiredCoresLocal -= (id)
                speed -= (id)
        }}
}

return correctedCores
```

nominalRateApp is calculated starting from the information that are present in the profiling file. Since it resides on the master machine, we needed to calculate its value there and then send it to the workers as part of the message `InitControllerExecutor` that is already used to

create the controllers of the various executors. For a matter of convenience, the calculation has been included in the `HeuristicBase` class in the method `loadConfigValues`. As we can see in Listing 4.21, we simply parse the profiling informations contained in the JSON and lookup for the *nominalRate* and the *weight* of those stages that will be executed, those whose field *skipped* is false. As we can see, the resulting *weightedNominalRate* is stored as a configuration variable, because we need to use it different times, every time we are willing to send the message to initialize an executor controller.

Listing 4.21: `HeuristicBase`: calculating the avg nominal rate of the application.

```
def loadConfigValues(appJson: JsValue): Unit = {
var weightSum: Double = 0
var weightedNominalRateSum: Double = 0

appJson.asJsObject.fields.keys.toList.foreach(id => {
        val stageJson = appJson.asJsObject.fields(id).asJsObject
        if (!stageJson.fields("skipped").convertTo[Boolean]) {
                val nominalRate = stageJson.fields("nominalrate")
                    .convertTo[Double]
                val weight = stageJson.fields("weight").convertTo
                    [Double]
                weightSum += weight
                weightedNominalRateSum += nominalRate * weight
        }
})

val weightedNominalRate = weightedNominalRateSum / weightSum
conf.set("spark.control.nominalrateapp", weightedNominalRate.
    toString)
}
```

The method is executed when a new application is launching, as shown in Listing 4.22 in `DagScheduler` class, after having checked that the deadline provided is feasible according to the profiling data, we calculate the nominal rate. The value is then retrieved in `ControlEventListener` and is used as argument of the previous mentioned message.

Listing 4.22: `DagScheduler`: calculating the average nominal rate of the starting application.

```
 if (appJson != null && sc.conf.getBoolean("spark.control.
     checkdeadline", false)) {
        logInfo("LOADED JSON FOR APP: " + jsonFile)
        if (!heuristic.checkDeadline(appJson)) {
                stop()
        }
        heuristic.loadConfigValues(appJson)
}
```

5 EVALUATION

In this chapter we describe the experiments that have been conducted in order to evaluate both the feasibility of controlling the execution of parallel Spark applications by tuning the number of CPU cores assigned to the their executor and xSpark itself.

Tests have been conducted on *Standard_D14_v2* [39] VM provided by Microsoft Azure [23], each of them has 16 vCPU, 112 GiB of memory, 800 GiB of local SSD storage and 6000 Mbps network bandwidth. This kind of VM is specifically optimized for memory usage, with an high memory-to-core ratio. Each machine is running Canonical Ubuntu Server 14.04.5-LTS [36], Oracle Java 8 [17], Apache Hadoop 2.7.2 [1], Apache Spark 2.0.2 [2] and xSpark. All the VM software is stored in 200 GiB virtual hard disk persistently kept in Azure Blob Storage [24]. The benchmark cluster is composed by 5 VM running HDFS and 5 running Apache Spark and xSpark.

In Table 5.1 is reported the configuration of xSpark, meanwhile Apache Spark is run with its out-of-the-box configuration when no particular settings are mentioned.

5.1 BENCHMARKS

Here we introduce the different benchmarking applications that have been used in order to evaluate the performances of xSpark and to compare it with Apache Spark.

5.1.1 *Spark-Bench*

Spark-Bench [22] is a benchmarking suite for Apache Spark[1]. It is composed by a set of different workloads belonging to four application types supported by Spark 2.0.x, including machine learning, graph processing, streaming and SQL queries. A data generator is provided in order to allow users to generate arbitrary size of input data. From this suite we used *KMeans* and *SVM*, from the Spark's machine learning library, and *PageRank*, from Spark's graph processing library.

In Table 5.2 is reported the configuration of Spark-Bench applications that have been used in our experiments.

1 Available at https://github.com/SparkTC/spark-bench/tree/legacy

PARAMETER	VALUE
spark.control.alpha	0.95
spark.control.beta	0.33
spark.control.heuristic	Control Unlimited
spark.control.tsample	1,000
spark.control.k	50
spark.control.ti	12,000
spark.control.corequantum	0.05
spark.control.maxexecutor	4
spark.control.coreforvm	16
spark.control.avgnominalrate	1,000,000

Table 5.1: xSpark configuration.

5.1.2 *Spark Performance Test*

Spark Performance Test[2] (Spark-Perf) is a performance testing framework for Apache Spark 1.0+. It covers Spark Core RDD, SQL, DataFrames and Machine Learning. We used this suite to test basic aggregation and sorting functionalities of Spark, in particular using *aggregate-by-key* and its variations.

In Table 5.3 are presented the configuration parameters of Spark-Perf applications that have been used in our tests.

5.1.3 *TPC Benchmark H*

Transaction Processing Performance Council (TPC) Benchmark™H [6] is a decision support benchmark[3]. It consists of a suite of business oriented queries and concurrent data modifications. The data and queries have been chosen in order to have broad industry-wide relevance. We discarded the performance metric reported by TPC-H, Composite Query-per-Hour, since we are only interested in the execution of the queries.

In Table 5.4 are reported the configuration settings that have been used when generating TPC-H dataset. TPC-H data generator generates 8 different tables with different sizes, as shown in Table 5.5. A *Scale Factor* equal to one generates 1 GiB of data, *Number of Partitions* is used to split large tables in the given number of partitions.

2 Available at https://github.com/databricks/spark-perf
3 Available at https://github.com/ssavvides/tpch-spark

APP	PARAMETER	VALUE
KMeans	Number of Partitions	1,000
	Number of Points	100,000,000
	Number of Clusters	10
	Dimensions	10
	Scaling	0.6
	Number of Iterations	1
SVM	Number of Partitions	1,000
	Number of Examples	100,000,000
	Number of Features	5
	Number of Iterations	1
PageRank	Number of Partitions	1,000
	Number of Vertexes	3,000,000
	μ	3.0
	Number of Iterations	1

Table 5.2: Spark-Bench configuration.

PARAMETER	VALUE
Scale Factor	5
Number of Partitions	400
Number of Trials	1
Number of Records	200,000,000
Number of Unique Keys	20,000
Dataset Type	String

Table 5.3: Spark-Perf configuration.

5.2 METRICS

We decided to test the multi-application performances of xSpark by running composite benchmarks. With composite benchmark we intent a set of different applications that can have different deadline and release time. We use as release time the delay of the launch of an application with respect to the first application launched. In this way, the first application launched will have delay zero, the second one that will start a certain amount of seconds later, will have this span of time as delay.

In order to evaluate the performances of the composite benchmarks, we took into account different metrics:

- #A is the number of applications that have completed in advance with respect to their assigned deadline

PARAMETER	VALUE
Scale Factor	40
Number of Partitions	64

Table 5.4: TPC-H configuration.

TABLE	SIZE
Part	$200,000 * ScaleFactor$
Part Supplier	$800,000 * ScaleFactor$
Supplier	$10,000 * ScaleFactor$
Customer	$150,000 * ScaleFactor$
Line Item	$6,000,000 * ScaleFactor$
Orders	$1,500,000 * ScaleFactor$
Nation	25
Region	5

Table 5.5: Tables and their size generated by *TPC-H* data generator.

- #D is the number of applications that have completed in delay with respect to their assigned deadline
- ϵ is the average deadline error, computed as the average of the deadline error of the applications that are inside a composite benchmark. It is important to remember that the deadline error is calculated as the absolute value of $\frac{Deadline - ExecutionTime}{Deadline}$, where execution time is the time needed to complete the execution of the application's tasks.
- ϵ_A is the sum of the deadline errors of the applications that have completed the execution before their deadline, in particular this value is zero if no applications have completed on time, i.e. when #A is zero
- ϵ_D is the sum of the deadline errors of the applications that have completed the execution after their deadline, in particular this value is zero if no applications have exceeded their expected execution time, i.e. when #D is zero
- ϵ_{abs} is the absolute sum of the deadline errors of the applications, intended as how much error we had in the execution

In order to validate the values we obtained from the different executions, we decided to examine how vanilla Spark would behave if it had an Earliest Deadline First like task scheduler across its applications, in particular trying to give all the available resources to a

single application, since we want to mimic the behavior of EDF "All" with $\gamma = 1$ that we have discussed in Section 3.6.1. We called this approach Clairvoyance EDF, since when performing the analysis we exactly know all the applications that will be submitted and the duration of all their tasks. To do so, we used Spark's event log in order to extract the different tasks and their duration. We also needed to take care of Spark's overhead (e.g., DAG submission times), in particular we distributed the overheads equally among all the tasks, in particular the overhead has been calculated as

$$\frac{Duration - ExeTimePerCore}{NumTasksPerCore}$$

where

$$ExeTimePerCore = \frac{SerializedExeTime}{NumExecutors * NumCoresPerExe}$$

and

$$NumTasksPerCore = \frac{NumTasks}{NumExecutors * NumCoresPerExe}$$

Duration is obtained by comparing the timestamps of the two events `SparkListenerApplicationStart` and `SparkListenerApplicationEnd` that represent the starting and the ending of a Spark application, in particular no computation is performed respectively before and after these two events. *SerializedExeTime* is the sum of the duration of the various Spark tasks, retrieved from the event `SparkListenerTaskEnd` that represent the completion of a task execution, in particular we compare the launch timestamp and the finish timestamp in order to understand the task's duration. *NumTasks* represent the count of the tasks that compose the application, *NumExecutors* is the number of running executors and *NumCoresPerExe* is the number of available cores in each of the executor.

In order to suggest an EDF scheduling of the various applications tasks, we begin assigning tasks to empty cores according to the deadlines of the applications. In particular we try to saturate the system resources with the executable tasks of an application, i.e., those which belongs to a stage that does not depend on a not already finished one. When no tasks are schedulable due to precedence constraints, we allocate tasks of another application according to its deadline.

This tasks allocation has the goal of checking if it is possible to execute the composite benchmark respecting the requested deadlines, since we want to be sure that no further overhead has been introduced in the application execution. It is important to remember that the Clairvoyance EDF results might not be exactly the same as the one obtained from a real execution, this is due to different factors such as network delays when launching applications, resource contention such as busy I/O devices, etc. Indeed, it has been used only to estimate a possible behavior of the composite benchmark, e.g. how many application will exceed their deadline.

BENCHMARK	APP	DELAY	DL	NR
Spark-Bench	PageRank	0	250	671999
	KMeans	40	160	5142187
	SVM	80	250	46898
Spark-Perf	ABK	0	120	319623
	ABK-naive	40	200	486746
	ABK-int	80	160	267094
TPC-H	TPCH_21	0	210	26717
	TPCH_22	30	147	3056
	TPCH_18	60	136.5	362703
	TPCH_9	90	115.5	3480
	TPCH_8	120	103.25	3703

Table 5.6: The three composite benchmark used to evaluate the performances of the different strategies with respect to different kind of applications. From left to right: composite benchmark (*Benchmark*), application (*App*), delay in seconds (*Delay*), deadline in seconds (*DL*), application nominal rate (*NR*). *ABK* stands for *aggregate-by-key*.

5.3 DIFFERENT KIND OF APPLICATIONS

In Table 5.6 we show the three composite benchmarks that we will analyze in the following subsections. We choose to perform one experiment for each of the benchmarking suite that we are using, in order to evaluate the performances of xSpark with different kind of applications. The proposed deadlines are feasible in the single application context, we want to investigate if they can also be satisfied when running multiple applications together. We want to inspect how the different strategies behave with respect to the different application types and the value of Supervisor γ. Each composite benchmark has been executed using all the strategies (EDF "All", EDF "Pure", EDF "Proportional", Proportional and Speed) and with Supervisor γ value 0 or 1.

The Tables 5.7, 5.8 and 5.9 contain the results of the composite bench regarding Spark-Bench, Spark-Perf and TPC-H applications respectively. *#A* and *#D* column represent the average number of applications that ended in advance and in delay, respectively. ϵ is the average deadline error, where the deadline error of an application is calculated as $\frac{\mathrm{Deadline}-\mathrm{ExecutionTime}}{\mathrm{Deadline}}$. ϵ_A and ϵ_D are the sum of the errors of the applications that ended in advance and those which delayed, respectively, meanwhile ϵ_{abs} is the sum of the two representing the absolute error.

STRATEGY	γ	#A	#D	ϵ	ϵ_A	ϵ_D	ϵ_{abs}
EDF "All"	0	3	0	12.41	37.24	0.00	37.24
EDF "All"	1	3	0	37.24	111.73	0.00	111.73
EDF "Pure"	0	3	0	6.63	19.89	0.00	19.89
EDF "Pure"	1	3	0	12.05	36.16	0	36.16
EDF "Prop."	0	3	0	6.72	20.16	0.00	20.16
EDF "Prop."	1	3	0	17.10	51.29	0.00	51.29
Proportional	0	2	1	6.88	7.59	13.07	20.66
Proportional	1	3	0	13.75	41.26	0	41.26
Speed	0	3	0	5.52	16.56	0	16.56
Speed	1	3	0	18.16	54.47	0.00	54.47

Table 5.7: Results of the composite benchmark concerning *Spark-Bench* applications: *PageRank*, *KMeans* and *SVM*. *Prop.* stands for *Proportional*.

5.3.1 *Spark-Bench composite benchmark*

In this composite benchmark we executed in order *PageRank*, *KMeans* and *SVM*, with a delay of 40 seconds between the launch of each of the applications. All the deadline proposed are feasible, indeed profiling execution times of the applications are respectively 60, 86 and 77 seconds. Each application executor uses 33 GiB of memory for heap allocation. Off-heap allocation is disabled.

As we can see from Table 5.7, in most of the configurations we achieve in completing the execution of the applications before the proposed deadline, indeed column *#A* value is 3 in 9 out of 10 cases.

First thing to notice is that when using γ equal to 1 instead of 0 with strategy Proportional, we increase the number of applications that end in advance, we can see that the number of applications that ends in delay (#D) is reduced from 1 to 0. It is important to remember that choosing $\gamma = 1$ is not always the best choice, first of all because in this experiment we always increase the total error of the applications that end in advance ϵ_A and as a consequence, the average deadline error ϵ, but also we need to consider that if there is only one application running in the cluster for its entire execution time, the result is that it will always acquire all the resources available. As a result, we will have a larger deadline error, for example if we consider EDF "All" we go from $\epsilon = 12.41$ with $\gamma = 0$ to $\epsilon = 37.24$ with $\gamma = 1$, which is about three times greater.

The smallest deadline errors in this experiment are achieved when using strategies EDF "Pure" ($\epsilon = 6.63$), EDF "Proportional" ($\epsilon = 6.72$),

STRATEGY	γ	#A	#D	ϵ	ϵ_A	ϵ_D	ϵ_{abs}
EDF "All"	0	2	1	20.27	39.55	21.25	60.81
EDF "All"	1	2	1	24.48	56.78	16.67	73.44
EDF "Pure"	0	1	2	10.86	5.83	26.75	32.58
EDF "Pure"	1	2	1	11.55	12.16	22.50	34.66
EDF "Prop."	0	1	2	11.07	5.83	27.38	33.21
EDF "Prop."	1	1	2	10.84	6.11	26.42	32.53
Proportional	0	1	2	12.63	3.89	34.00	37.89
Proportional	1	1	2	11.75	5.00	30.25	35.25
Speed	0	1	2	13.68	5.55	35.50	41.05
Speed	1	1	2	11.50	5.83	28.67	34.50

Table 5.8: Results of the composite benchmark concerning Spark-Perf applications: *aggregate-by-key*, *aggregate-by-key-naive* and *aggregate-by-key-int*. *Prop.* stands for *Proportional*.

Proportional ($\epsilon = 6.88$) and Speed ($\epsilon = 5.52$), with Supervisor $\gamma = 0$. Although, we need to consider that choosing strategy Proportional has the effect of having 1 out of 3 applications ending delayed (#D = 1). This might not be a problem if we want to ensure a fair distribution of resources across the application, at the price of slightly violating some of the deadlines. Otherwise, by choosing the other three strategies will result in having zero violations in this test scenario.

By analyzing the same composite benchmark with the Clairvoyance EDF method previously introduced, we obtain that all the applications should respect the proposed deadline (#A = 3). In particular we obtain that *PageRank* should complete with a deadline error of 35.31, *KMeans* with 45.99 and *SVM* with 36.57, this results in having $\epsilon = 39.29$, $\epsilon_A = \epsilon_{abs} = 117.87$ and $\epsilon_D = 0$. The real EDF "All" with $\gamma = 1$ execution completes the same number of applications in advance and the obtained errors are 5% smaller.

5.3.2 *Spark-Perf composite benchmark*

In this experiment we execute three different kind of aggregations from the Spark-Perf benchmarking suite. We executed in order *aggregate-by-key*, *aggregate-by-key-naive* and *aggregate-by-key-int*, with a delay of 40 seconds between each of the applications. Even though all the applications are performing aggregation, due to the different implementation we can notice different application nominal rates, which can be seen in Table 5.6. The execution time that have been measured during the profiling phase are respectively 61, 78 and 102 seconds, which

are smaller than the chosen deadlines. Each application executor uses 33 GiB of memory for heap allocation and off-heap allocation is disabled.

From Table 5.8 we can notice that we never achieve in satisfying all the deadlines. In particular, when using EDF "All" we only satisfy 2 out of 3 deadlines and in general this strategy is the one that minimizes the number of violations, in particular when paired with Supervisor $\gamma = 1$.

If we are running this composite benchmark in a situation in which we have a "strict" deadline, which means the penalty paid when we break the deadline is infinite, we need to minimize the number of violations. As a result, we need to choose for example those strategy whose value in the column *#D* is the smallest (#D $= 1$), such as EDF "All" or EDF "Pure" with $\gamma = 0$

Instead, if the deadline is a "soft" boundary, which means paying a penalty that is proportional to the delay errors, we want to minimize the value of ϵ_D. This is done another time by choosing EDF "All" and $\gamma = 1$ ($\epsilon_D = 16.67$).

If we disregard the concept of violating the deadline, we may want to minimize the average deadline error ϵ, in this composite benchmark scenario the best choice is to use EDF "Proportional" with $\gamma = 0$ ($\epsilon = 11.07$) and $\gamma = 1$ ($\epsilon = 11.75$) or EDF "Pure" with $\gamma = 0$ ($\epsilon = 10.86$) or Speed with $\gamma = 1$ ($\epsilon = 11.50$).

Analyzing the same composite benchmark with the Clairvoyance EDF method previously discussed, we obtain that only 2 out of 3 applications succeed in completing before the designated deadline (#A $= 2$ and #D $= 1$). In particular, *aggregate-by-key* and *aggregate-by-key-naive* succeed in completing before the deadline with a deadline error that is respectively 25.13 and 7.57, *aggregate-by-key-int* instead fails in completing on time with an error of 54.58. As a result, we obtain $\epsilon = 29.09$, $\epsilon_A = 32.70$, $\epsilon_D = 54.58$ and $\epsilon_{abs} = 87.28$. Real execution of EDF All with $\gamma = 1$ achieves in completing the same number of applications in time, with ϵ and ϵ_{abs} that are about 15% smaller.

5.3.3 *TPC-H composite benchmark*

In order to test SQL queries performances in xSpark, we decided to use queries from the benchmarking suite TPC-H. After having profiled all the 22 queries proposed, we choose to use only the top 5 queries with respect to their execution times.

These queries are:

Q8 National Market Share. This query determines how the market share of a given nation within a given region has changed over two years for a given part type.

STRATEGY	γ	#A	#D	ϵ	ϵ_A	ϵ_D	ϵ_{abs}
EDF "All"	0	3	2	38.17	133.31	57.53	190.85
EDF "All"	1	3	2	39.15	141.49	54.26	195.75
EDF "Pure"	0	3	2	23.97	62.33	57.54	119.87
EDF "Pure"	1	3	2	20.04	64.61	35.60	100.21
EDF "Prop."	0	3	2	14.86	41.79	32.52	74.32
EDF "Prop."	1	3	2	15.66	47.59	30.72	78.31
Proportional	0	1	4	15.18	22.68	53.24	75.91
Proportional	1	2	3	11.31	29.73	26.81	56.53
Speed	0	1	4	12.73	3061	33.06	63.67
Speed	1	2	3	10.05	37.98	12.29	50.27

Table 5.9: Results of the composite benchmark concerning TPC-H queries: *Q21*, *Q22*, *Q18*, *Q9* and *Q8*. *Prop.* stands for *Proportional*.

Q9 Product Type Profit Measure. This query determines how much profit is made on a given line of parts, broken out by supplier nation and year.

Q18 Large Volume Customer. This query ranks customers based on their having placed a large quantity order. Large quantity orders are defined as those orders whose total quantity is above a certain level.

Q21 Suppliers Who Kept Orders Waiting. This query identifies certain suppliers who were not able to ship required parts in a timely manner.

Q22 Global Sales Opportunity. This query identifies geographies where there are customers who may be likely to make a purchase.

The queries are in order *Q21*, *Q22*, *Q18*, *Q9* and *Q8*, with a delay of 30 seconds between each of them. All the deadlines requested are larger than the execution time measured in the profiling phase, that is respectively 120, 84, 78, 66 and 59 seconds. Each application executor uses 20 GiB of memory for heap allocation. Off-heap allocation is disabled.

As we can see from Table 5.9, also this time we did not achieve in satisfying all the deadline in any of the proposed configurations. Indeed EDF "All" with $\gamma = 1$ only satisfies 3 out of 5 requested deadlines, and this is the strategy whose best effort is to minimize the number of violations (#D $= 2$). In this case, this strategy also maximizes the total error in advance ($\epsilon_A = 141.71$).

If we want to minimize the average deadline error ϵ, our choice can be among Proportional strategy with $\gamma = 1$ ($\epsilon = 11.31$) or Speed with $\gamma = 0$ ($\epsilon = 12.73$) or $\gamma = 1$ ($\epsilon = 10.05$). Notice that choosing Speed with $\gamma = 1$ the result is that we also minimize the total delay error ($\epsilon_D = 12.29$), this can be a good point if we are in a scenario in which the provided deadlines are soft boundaries, in which the penalty paid for violating them grows with the delay error.

Instead if the deadline is "strict", meaning that an infinite penalty is paid, we can choose among the strategies that achieve in completing the same number of application of EDF "All" ($\#A = 3$), that are EDF "Pure" and EDF "Proportional" with Supervisor $\gamma = 0$ or $\gamma = 1$.

Analyzing the same benchmark with the Clairvoyance EDF approach that has been proposed, we obtain that we should be able to complete 3 out of 5 applications on time. The method suggests that queries 22, 18 and 9 should complete on time with a deadline error that is respectively of 82.95, 41.55 and 0.72. The other two queries 21 and 8 are not able to satisfy the deadline, and have an error of 31.99 and 90.86. The result is having a $\epsilon = 49.614$, $\epsilon_A = 125.22$, $\epsilon_D = 122.85$ and $\epsilon_{abs} = 248.07$. Real execution of EDF All with $\gamma = 1$ achieves in completing the same number of applications in time, with ϵ and ϵ_{abs} that are about 20% smaller.

5.3.4 *Which configuration to choose?*

Sadly this is not an easy question to answer. In order to use xSpark with multiple applications, we need to select a contention resolution strategy and the value of the parameter $\gamma \in [0, 1]$.

Starting from the Supervisor γ parameter, we need to remember that setting its value equal to 1 will have the effect that the cluster CPU resources will be fully used even though the applications running at a certain moment might need less resources than the ones they will end up having allocated. Selecting value 0 instead has the effect of allocating only the requested cores in order to keep up with the desired progress rate, if this is possible in the sense that the total sum of the requested CPU cores is lower than the system maximum number.

Selecting $\gamma = 1$ is not a problem if on average we have more than one application running and the requested deadlines are shorter, since in this way we allow applications that are running alone at a certain time to speed up their execution by allocating more cores, in order to better tolerate the future presence of other applications. Instead, if it happens often that an application spends most of its execution time alone in the cluster, using $\gamma = 1$ might not be the best choice, since the application will acquire more resources than the needed ones and complete its execution possibly much time ear-

lier with respect to the desired deadline. In this case, we can choose to use $\gamma = 0$.

By analyzing the results of the previous composite benchmarks, we see that choosing $\gamma = 1$ instead of $\gamma = 0$ made us improve the number of applications completed in advance (#A) in 4 cases out of 11 where $\gamma = 0$ did not achieved in completing all the executions before the deadline. When $\gamma = 0$ already succeeds in satisfying the requested deadlines, as in the Spark-Bench composite benchmark discussed in Section 5.3.1, selecting $\gamma = 1$ has the result of increasing the deadline error ϵ, this because it increments the total error of the applications that end in advance ϵ_A, in particular it is between 2 and 3 times the previous value.

On the contention resolution strategy side, we can choose among EDF "All", EDF "Pure", EDF "Proportional", Proportional and Speed.

EDF "All" has the purpose of allocating all the cluster resources to the application whose deadline is the closest. As a result, we try to complete the selected application as fast as possible, by possibly pausing the execution of other applications. The result of this approach is that we maximize the error of the applications that end in advance ϵ_A and the number of these applications #A, as we have already shown in the previous composite benchmarks. This strategy has been mostly used to understand if the proposed combination of application deadlines are feasible together, indeed if they are not all satisfiable using EDF "All", they are not even using other strategies.

EDF "Pure" allocates the requested resources to the executors according the priority of their application, that is determined by their remaining time to complete, that is positive when they have not exceeded their deadline. According to the composite benchmarks executed, in 5 out of 6 cases the number of applications completed in advance is the same as the ones done by EDF "All". The same applies for EDF "Proportional", which allocates resources using an application weight that is proportional to their remaining time to complete, where in all the cases the number of applications completed in advance is the same as the one of EDF "All". The main difference is that the average deadline error of both EDF "Pure" and EDF "Proportional" is halved with respect to the one of EDF "All".

Proportional and Speed strategy employ a different way to repartition resources among the applications, the former assigns CPU cores in a way that is proportional to the requested ones to keep up with the desired progress rate, the latter inversely proportional to the application nominal rate. In the previously discussed experiments, both the strategy have an average deadline error that is below the average for each of the composite benchmark. For example, considering TPC-H composite benchmark discussed in Section 5.3.3, the average deadline error ϵ is 20.98 and 19.24 respectively for $\gamma = 0$ and $\gamma = 1$,

Proportional (15.18 and 11.31) and Speed (12.73 and 10.05) strategy errors instead are almost the half.

In Section 6 we will introduce different use cases and explain which combination of strategy and Supervisor γ value we may choose to satisfy our interests.

5.4 DEADLINE AS A PRIORITY BETWEEN APPLICATIONS

When running on a cluster, each Spark application gets an independent set of executors, that run tasks and store data only for a given application. The same thing is true for xSpark.

The main difference is that, when we run Spark in standalone mode, by default, all applications submitted will run in FIFO order, each of them will try to use all the available nodes. We can control the static partitioning of resources by changing the application allocated cores and memory using Spark properties `spark.cores.max` and `spark.executor.memory`.

In the next experiments, we compared the execution of Spark and xSpark. The idea is that we have a set of applications that are submitted delayed of a certain amount of time. Every application has a different deadline, with the meaning of desired completion time. We want that the execution of the application is completed before the deadline, given that it is a feasible request.

The expected behavior is that Spark will execute the applications in a FIFO style, allocating all the cluster resources for every application at a time. xSpark instead will parallelize the execution of the applications, in this way it can use the deadline as a way to give priority to a particular application with respect to another.

5.4.1 *Comparison 1: Spark-Bench*

In this experiment, we executed three different applications from Spark-Bench in Apache Spark and xSpark.

As shown in Table 5.10, we executed *KMeans*, *PageRank* and *SVM* with a delay of 40 seconds between the starting of the first application and the second one, and 60 seconds between the second and the third ones. All the deadlines are feasible, indeed they are all greater than the execution time we observed when profiling the applications. In Spark, the executors will reserve 100 GiB for their JVM, instead in xSpark they will only allocate 33 GiB in order to be able to run safely without incurring in memory shortage problems. Off-heap allocation is not enabled. In both the tests, applications can use 15 CPU cores.

In order to compare the results of Spark and xSpark, we tested three different configurations: i) Spark 2.0.2, ii) xSpark with contention strategy EDF "All" and Supervisor $\gamma = 1$, iii) xSpark with contention

APP	DELAY	DEADLINE
KMeans	0	300
PageRank	40	300
SVM	100	120

Table 5.10: Spark-Bench applications (*App*) run in Comparison 1 with delay (*Delay*) and deadline (*Deadline*) expressed in seconds.

CONFIG	APP	DL	ET	ERR	MISSED
Spark	PageRank	300.0	58.964	80.3%	No
	KMeans	300.0	105.406	64.7%	No
	SVM	120.0	142.573	18.8%	Yes
EDF **"All" 1**	PageRank	300.0	74.846	75.1%	No
	KMeans	300.0	190.545	36.5%	No
	SVM	120.0	81.248	32.3%	No
EDF **"Pure" 0**	PageRank	300.0	289.059	3.6%	No
	KMeans	300.0	284.654	5.1%	No
	SVM	120.0	116.307	3.1%	No

Table 5.11: Spark-Bench applications results run in Comparison 1. From left to right: configuration used (*Config*), application (*App*), deadline in seconds (*DL*), execution time in seconds (*ET*), deadline error (*Err*), missed deadline (*Missed*).

strategy EDF "Pure" and Supervisor $\gamma = 0$. These executions have been repeated three times to have more accurate results

In Table 5.11 we compare the results of the different configuration results. The execution time (ET) has been calculated from the Spark events log file, in particular as the difference between the timestamps of the events `SparkListenerApplicationStart` and `SparkListenerApplicationEnd`. The error (ERR) has been calculated as $\frac{DL-ET}{DL}$ where DL is the requested deadline.

Due to the FIFO scheduling of the applications in Apache Spark and the fact that every application allocates all the resources in the cluster, we can see that the deadline requested for *SVM* cannot be satisfied. Moreover, from Figure 5.1a we can easily visualize that the effective execution of the last application begins when the deadline is almost expired, since it needs to wait until the previous application has released all the resources. This because Spark has no built in concept of deadline, but even supposing the existence of an application scheduler in Spark that is not FIFO but that selects the pending application whose deadline is closer, the situation would not have changed. Indeed when *KMeans* can launch its executor, *SVM* has yet

APP	DELAY	DEADLINE
Aggregate-by-key	0	300
Aggregate-by-key-naive	40	320
Aggregate-by-key-int	100	120

Table 5.12: Spark-Perf applications (*App*) run in Comparison 2 with delay (*Delay*) and deadline (*Deadline*) expressed in seconds.

to be submitted to the cluster, so the only application pending is the one that will start.

Using xSpark instead, allows us to exploit the provided deadline and satisfy all the three proposed deadlines in both the configuration proposed.

Choosing EDF "All" strategy with Supervisor $\gamma = 1$ allows us to terminate the execution of the composite test satisfying all the deadlines, as we can see in Figure 5.1b. However, due to the nature of this strategy, we have high deadline errors, in Table 5.11 we can see that *PageRank* completes a deadline error of 75.1%.

Selecting EDF "Pure" strategy and Supervisor $\gamma = 0$ (Figure 5.1c) we increase the execution time of each application with respect to the other xSpark configuration, but we obtain a lower deadline error, indeed all the applications terminate the execution with a deadline error that is smaller than 5%, as it is reported in Table 5.11.

5.4.2 *Comparison 2: Spark-Perf*

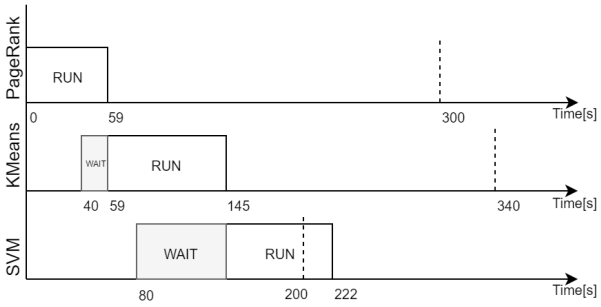
This experiment is similar to the previous one, but we used Spark-Perf applications instead of the Spark-Bench ones.

Deadlines proposed in Table 5.12 are feasible and larger than the execution times measured during the profiling phase. As in the previous experiment, Spark executor will allocate 100 GiB of memory meanwhile xSpark ones only 33 GiB, without using off-heap allocation, both of them will use up to 15 cores during the execution.

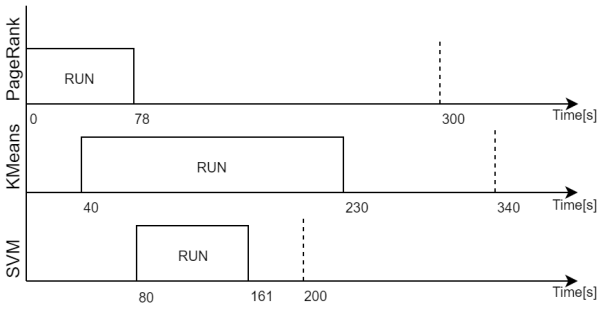
In order to compare the results of Spark and xSpark, we tested four different configurations: i) Spark 2.0.2, ii) xSpark with contention strategy EDF "All" and Supervisor $\gamma = 1$, iii) xSpark with contention strategy EDF "Pure" and Supervisor $\gamma = 0$. These executions have been repeated three times to have more accurate results

In Table 5.13 results of the different configurations tested are presented. ET is the execution time and has been calculated as explained in the previous subsection. Same thing applies for the error ERR.

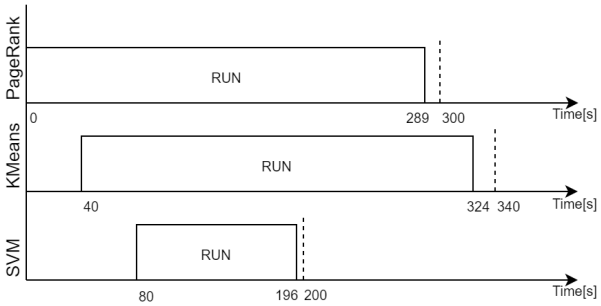
As in the previous experiment, we notice that Apache Spark is not able to satisfy all the requested deadlines, indeed last launched application completes its execution more than 20 seconds after the requested deadline. This can be easily seen in the diagram shown in



(a) Spark

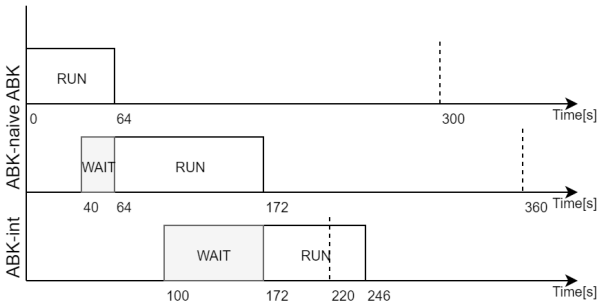


(b) EDF "All" 1

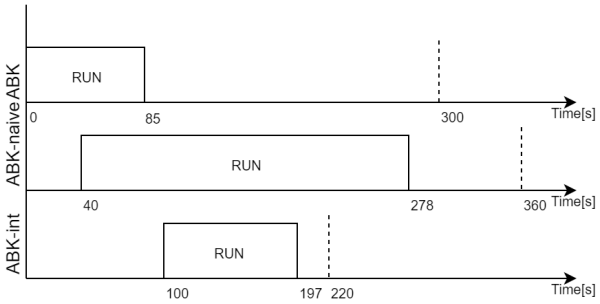


(c) EDF "Pure" 0

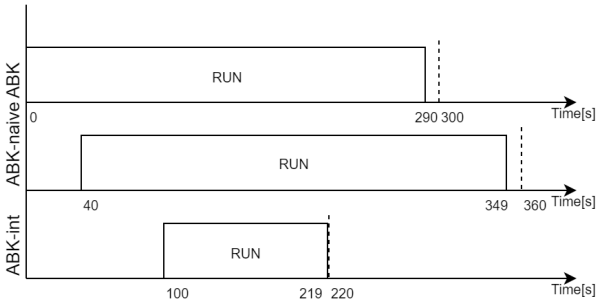
Figure 5.1: Diagrams that show the execution of the applications in Comparison 1. The dotted segment represent the deadline for the given application. The rectangle representing the application is *gray* when it is waiting the presence of enough resources to be scheduled, *white* when running.



(a) Spark



(b) EDF "All" 1



(c) EDF "Pure" 0

Figure 5.2: Diagrams that show the execution of the applications in Comparison 2. The dotted segment represent the deadline for the given application. The rectangle representing the application is *gray* when it is waiting the presence of enough resources to be scheduled, *white* when running.

CONFIG	APP	DL	ET	ERR	MISSED
Spark	ABK	300.0	64.524	78.5%	No
	ABK-naive	320.0	132.493	58.6%	No
	ABK-int	120.0	146.652	22.2%	Yes
EDF **"All" 1**	ABK	300.0	85.169	71.6%	No
	ABK-naive	320.0	238.312	25.5%	No
	ABK-int	120.0	97.349	18.9%	No
EDF **"Pure" 0**	ABK	300.0	290.792	3.0%	No
	ABK-naive	320.0	309.691	3.2%	Yes
	ABK-int	120.0	119.755	0.2%	No

Table 5.13: Spark-Perf applications results run in Comparison 2. From left to right: configuration used (*Config*), application (*App*), deadline in seconds (*DL*), execution time in seconds (*ET*), deadline error (*Err*), missed deadline (*Missed*). *ABK* stands for *aggregate-by-key*.

Figure 5.2a. As previously explained, this is due to the FIFO application scheduler and the static resource allocation that by default standalone Spark uses.

Using xSpark strategy EDF "All" with Supervisor $\gamma = 1$ we try to avoid deadline violation caused done by *aggregate-by-key-int*. Indeed the diagram in Figure 5.2b shows that all the three applications achieve the result of completing before the desired deadline.

Our goal now is to reduce the error with respect to the requested deadlines, in particular in the previous strategy analyzed we have an high error for the first application launched, 71.6% (from Table 5.13).

As in the previous experiment, we analyze the result using strategy EDF "Pure" and Supervisor $\gamma = 0$. From the diagram in Figure 5.2c we can see that using this strategy, we have no deadline violation, and all the applications complete their executions really close to the deadline. From results in Table 5.13 we can see that the deadline errors are very small (column *Err*), with the highest one being 3.2% from *aggregate-by-key-naive*.

5.4.3 *Comparison extension*

Parallelizing vanilla Spark with static allocation

One might object that it is obvious that a parallel execution outperforms a serialized one. To address this objection, we configured Spark in order to execute using a static allocation of resources, in particular each application will launch with one executor per worker node that has 33 GiB of memory and 5 CPU cores allocated. This prevents the applications from waiting the ending of the previously launched

CONFIG	APP	DL	ET	ERR	MISSED
Static	PageRank	300.0	86.073	71.3%	No
	KMeans	300.0	186.206	37.9%	No
	SVM	120.0	131.018	9.2%	Yes
Static	ABK	300.0	169.337	43.6%	No
	ABK-naive	320.0	256.841	19.7%	Yes
	ABK-int	120.0	215.762	79.8%	Yes

Table 5.14: Results of applications run using a static partitioning of resources in vanilla Spark. From left to right: configuration used (*Config*), application (*App*), deadline in seconds (*DL*), execution time in seconds (*ET*), deadline error (*Err*), missed deadline (*Missed*).

ones, instead they can immediately begin the computation without wasting time.

In order to obtain this desired behavior, we need to set `spark.executor.memory` to 33 GiB, representing the memory allocated to each of the executors, `spark.executor.cores` to 5, that represents the number of cores assigned to each of the executors, `spark.cores.max` to 20, in order to allocate 4 executors for each of the applications.

In Table 5.14 we can see the different performances achieved by the two composite benchmarks used in the previous sections. The first composite benchmark (PageRank, KMeans, SVM) executed in parallel completes in 226 seconds, that is comparable with the one of Spark FIFO, 222 seconds, and the one of xSpark with EDF "All" and $\gamma = 1$, 230 seconds. However, parallelized Spark is not able to satisfy all the deadlines, in particular it violates the one assigned to *SVM*. Using parallelized Spark in the second composite benchmark has the effect of increasing the benchmark completion time to 315 seconds, which is an increment of the 28% with respect to FIFO Spark execution. As expected, parallelized Spark is still not able to satisfy all the deadlines, in particular an error of 79.8% is obtained with the missed deadline of *aggregate-by-key-int*.

Dynamic off-heap in Spark

A further objection that is possible to make is that we are running the test knowing exactly the number of applications that will run in parallel. This might not be always true case, since it depends on the context where xSpark will be deployed. In order to solve this point, we re-executed the two composite benchmark changing the configuration of xSpark, in particular we choose to enable the use of dynamic off-heap memory and allocate a small amount of static heap memory. In particular, each executor will now use only 10 GiB of static heap memory. Given the size of the virtual machines we

CONFIG	APP	DL	ET	ERR	MISSED
EDF **"All" 1**	PageRank	300.0	91.867	69.4%	No
	KMeans	300.0	201.915	32.7%	No
	SVM	120.0	87.341	27.20%	No
EDF **"Pure" 0**	PageRank	300.0	290.393	3.2%	No
	KMeans	300.0	285.421	4.9%	No
	SVM	120.0	115.525	3.7%	No
EDF **"All" 1**	ABK	300.0	91.665	69.4%	No
	ABK-naive	320.0	266.937	16.6%	No
	ABK-int	120.0	104.962	12.5%	No
EDF **"Pure" 0**	ABK	300.0	290.791	3.1%	No
	ABK-naive	320.0	309.805	3.2%	No
	ABK-int	120.0	119.546	0.4%	No

Table 5.15: Results of applications run using off-heap allocation in xSpark. From left to right: configuration used (*Config*), application (*App*), deadline in seconds (*DL*), execution time in seconds (*ET*), deadline error (*Err*), missed deadline (*Missed*).

are using, we say that we are now testing the two benchmarks in a xSpark cluster that supports up to 5 applications running together. It is important to limit the number of applications that can run together because if we saturate the available memory with the heap of the executors, no free memory will be available for off-heap allocation.

In Table 5.15 we see the results of running xSpark using dynamic off-heap allocation. Comparing them with the ones explained in Section 5.4.1 and 5.4.2, we see no differences in terms of number of violated deadlines. Notice that this is not a general rule, indeed if the deadlines were stricter, it could be possible that some of them would no longer be satisfiable. This happens because accessing off-heap memory is slower than accessing the on-heap one, and thus the execution times of the applications increase.

The first composite benchmark completion time, when using EDF "All", increases of the 5%, from 230 seconds to 242 when compared to the results previously obtained. The application whose execution time increased most is *PageRank* with an increment of the 22%. It is important to notice that this big increment is not completely due to the use of off-heap memory allocation. This increment has been mainly caused by the submission of another Spark application whose priority is higher, i.e., its deadline is closer, before the completion of *PageRank*. The supervisor, which using EDF "All" strategy, will correctly assign all the resources to this new application. This can be easily spotted by analyzing Figure 5.3. After 80 seconds of execution, we see that the cores allocated to *PageRank* (blue line) drop to zero, due to the

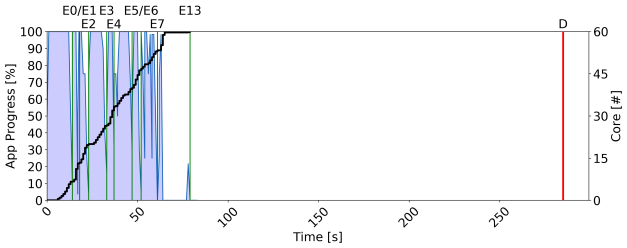


Figure 5.3: Cores allocated to PageRank application running in composite benchmark from Comparison 1 with off-heap memory allocation enabled.

starting of the aforementioned application. As soon as free cores are available, *PageRank* is able to complete its execution.

There are no relevant differences in the results of the executions using strategy EDF "Pure" with and without using off-heap allocation. Since deadlines are not strict, we are able to keep deadline errors below the 5% also when using off-heap allocation.

Comparing the executions of the second composite benchmark, with and without using off-heap allocation, we see that there is an increment around the 10% in the execution times when using strategy EDF "All". This increment is entirely due to to the nature of off-heap allocation, since now the sequence of "highest priority application" does not change when using off-heap allocation. In this experiment, the sequence is:

1. *aggregate-by-key*, when the benchmark starts
2. *aggregate-by-key-naive*, when *aggregate-by-key* has already completed and *aggregate-by-key-int* has not started yet
3. *aggregate-by-key-int*, when this application starts
4. *aggregate-by-key-naive*, when *aggregate-by-key-int* completes

The higher execution times have the effect of increasing the completion time of the composite benchmark, i.e., going from 278 seconds to 306 seconds when using off-heap allocation, which translates into a 10% increment.

If we compare the executions using strategy EDF "Pure", we se no relevant differences even in this second composite benchmark. Deadline errors are still under the 3% using off-heap memory allocation.

The use of dynamic allocation allows us to have a better utilization of the system memory. In particular, in the original experiment where we were using only a static allocation of one third of the total memory per application, the result was that it was possible that up to 66% of memory was unusable, because reserved for possible future applications. Using dynamic off-heap memory allocation instead, we can possibly have a 100% memory usage even when running a single

application. This is a great improvement in terms of memory utilization and flexibility, at the price of having slightly greater execution times.

5.5 OFF-HEAP AND HEAP PERFORMANCES

In this section we first inspect the impact of using different configuration of heap and off-heap memory allocation with respect to a single running application. Later on, we will compare the performances of the execution of multiple applications under different memory configurations.

5.5.1 *Heap and off-heap impact*

In this experiment, we compared the performances of two different applications: *Aggregate-by-key-naive* and *PageRank*, in particular for the last one we tested two different ways to store its RDD.

In Table 5.16 we can see the different configuration used, in particular we choose three distributions of heap and off-heap memory: all heap (100/0), balanced heap and off-heap (50/50), almost all off-heap (10/90). As previously introduced *PageRank* test was run using two different *Storage Levels*, "MemoryAndDisk" and "OffHeap". Each Storage Level records whether to use memory or External Block Store, whether to drop th RDD to disk if it falls out of memory or External Block Store, whether to keep the data in memory in a serialized format and whether to replicate the RDD partitions on multiple nodes. "MemoryAndDisk" uses memory and disk storage, storing data in a deserialized way, "OffHeap" instead allows also to store da in off-heap memory.

As shown in the previous table, we also tested the impact of using an ad-hoc profiling instead of using the basic one, intended as the one obtained from running the application using all the memory allocated in the heap.

In order to evaluate the differences among the different memory configurations and profiling file usage, we use three different metrics. $\mathtt{CpuTime}$ that represents the total amount of CPU seconds that have been used during the application execution. $\mathtt{Throughput} = \frac{\mathtt{NumberOfTasks}}{\mathtt{Duration}}$ that is higher when application completes in less time, since the $\mathtt{NumberOfTasks}$ is always the same given the same input data. $\mathtt{DeadlineError} = \frac{\mathtt{Deadline-ExecutionTime}}{\mathtt{Deadline}}$ that shows how precisely we can complete the execution of the application.

The first thing we can notice is that when we use an ad hoc profiling (see column *Prof*), we always have an higher deadline error (column *DE*) with respect to the case in which we use the profiling obtained from running the application using all the resources. So we have no

APP	MEM	PROF	CPUTIME	THR	DE
ABK-naive	100/0	Yes	5166	21.09	5.17
	50/50	No	5502	21.09	5.17
	50/50	Yes	5552	21.16	5.5
	10/90	No	6310	21.09	5.17
	10/90	Yes	6289	21.13	5.33
PageRank MD	100/0	Yes	1625	39.19	8.13
	50/50	No	1663	39.19	8.13
	50/50	Yes	1626	39.53	8.93
	10/90	No	2679	35.66	2.93
	10/90	Yes	2412	37.92	5.07
PageRank OH	100/0	Yes	2013	38.24	5.87
	50/50	No	1988	38.24	5.87
	50/50	Yes	2051	38.74	7.07
	10/90	No	2116	38.24	5.87
	10/90	Yes	2150	37.92	6.93

Table 5.16: Configurations used to inspect the impact of using different memory allocations and their results. From left to right: application (*App*), memory allocation used (heap / off-heap) in GiB (*Mem*), use of an ad hoc profiling (*Prop*), CPU time (*CpuTime*), throughput (*Thr*), deadline error (*DE*). *ABK* stands for *aggregate-by-key*, *MD* for *MemoryAndDisk*, and *OH* for *OffHeap*.

incentives in profiling an application against different memory allocations.

For both *aggregate-by-key-naive* and *PageRank "MemoryAndDisk"* we can see that when decreasing the memory allocated to the heap of the executors, the value of the CpuTime needed to complete the execution of the application increases (see column *CpuTime*). In particular, by analyzing the log files of *PageRank* running with 10 GiB allocated to the JVM heap we noticed that some of the blocks were needed to be dropped to disk. Disk swapping can be a serious threat to the performances of Spark applications. Due to the storage level select, *PageRank* blocks needed to be either persisted on memory or disk. A similar situation happened with *aggregate-by-key-naive*, with the only difference that in this case RDD blocks were not forced to be persisted.

Concerning *PageRank*, a solid improvement can be found by preventing disk swapping of RDD block. This can be easily done by selecting "OffHeap" storage level, in this way we can see that we reduce the impact of disk swapping on *CpuTime*. In particular in this experiment we reduce the *CpuTime* of *PageRank* running with 10 GiB allocated to the heap from 2679 to 2116 (without ad hoc profiling), which is a reduction around the 20%. Remember that all saved *CpuTime* can

METRIC	OH	25	20	15	10	5
Avg CPUT	No	5475	5994	6691	6741	6882
	Yes	5442	5770	5751	6781	6562
Avg DE	No	9.49	13.9	19.7	22.6	25.7
	Yes	8.6	11.5	10.7	21.3	21.5
Avg ET	No	172.2	174.0	183.2	193.7	203.9
	Yes	170.9	168.1	170.0	190.1	195.2
Avg THR	No	28.75	28.10	26.99	26.50	25.52
	Yes	29.01	28.78	28.76	26.26	26.30

Table 5.17: Results of executing Spark-Perf composite benchmark with different memory configuration. From left to right: the considered metric (*Metric*), dynamic off-heap enabled (*OH*), results with 5 different heap sizes in GiB (*25*, *20*, *15*, *10* and *5*). According to the metrics, *Avg CPUT* is the average CPU Time, *Avg DE* is the average deadline error, *Avg ET* is the average execution time, *Avg THR* is the average throughput.

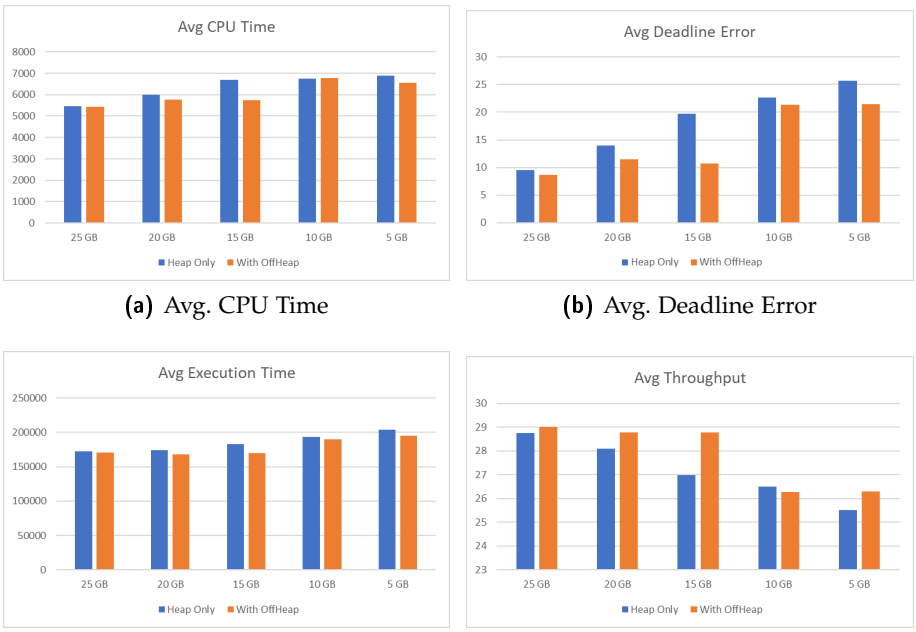
be used to schedule other applications, which is relevant in a multi application context.

5.5.2 *Multiple application*

We decided to inspect the effect of off-heap allocation when executing multiple applications, in particular we tested how Spark-Perf composite benchmark (Section 5.3.2) behaves under different memory conditions. In particular, we tested it using strategy Proportional with $\gamma = 0$, since we were interested only in understanding the effect of the different memory configurations. To perform our tests, we executed the composite benchmark using different values for the executor heap (i.e., 25, 20, 15, 10 and 5 GiB per executor) and repeated them enabling also dynamic off-heap allocation.

In Table 5.17 we can see the results obtained by executing the composite benchmark based on Spark-Perf applications with the different combinations of heap and off-heap memory. The metrics that we are analyzing are *Avg CPUT* that represents the average CPU time of the applications that compose the benchmark, *Avg DE* that is the average deadline error, *Avg ET* that is the average execution time, *Avg THR* that is the average throughput, calculated as $\frac{\mathrm{NumTasks}}{\mathrm{ExecutionTime}}$.

Plotting these values in Figure 5.4 helps us to visualize the effect of using different sizes of heap memory and the possible advantage of enabling off-heap allocation. From the diagrams we can easily understand the effect of reducing the size of the statically allocated heap, in particular by reducing the amount of memory allocated to the JVM heap of each executor we notice that the average CPU time increases



(a) Avg. CPU Time

(b) Avg. Deadline Error

(c) Avg. Execution Time

(d) Avg. Throughput

Figure 5.4: These diagrams show how choosing a different memory allocation in terms of heap and off-heap memory influences different metrics. In particular, 5 different heap size have been chosen and tested with and without dynamic off-heap enabled.

(Figure 5.4a), with a 25% increment between the minimum and the maximum value. As a result, also the other metrics are incremented, in particular the average deadline error grows from 9.49 when allocating 25 GB to 25.7 when using only 5 GB instead (Figure 5.4b), average execution time and throughput grow respectively of the 18% and 11% (Figure 5.4c and 5.4d). From the same set of graphs, we can see that enabling off-heap allocation has the result in general of improving all the metrics, indeed we a slightly lower CPU time (i.e., −5% on average) and execution time (i.e., −3%), remarkably lower deadline error due to the shorter duration (i.e., −18%) and slightly higher throughput (i.e., +2%).

It is important to remember that in general the effect of using off-heap allocation is positive in case of memory shortage, since storing values on off-heap memory is slightly slower than on-heap storage, but surely faster than writing and accessing data on disk.

5.6 THREATS TO VALIDITY

5.6.1 *Internal Threats*

The experiment were conducted launching composite benchmarks, composed by a set of applications, each with its own delay and dead-

line. We can assume that the chosen deadline has been set correctly for each of the applications, since it is passed as a configuration parameter to xSpark. Unluckily, the delay with which an application has been launched might not be the one we will expect, this is due to the tool that we are using to launch experiments and the benchmarking applications themselves. When the tool launches an application, it configures the parameters of the application and then launches it. Network delays might slow down this process, and thus having a resulting delay that is not always the same comparing different executions. In the executed experiment, we did not notice such a behavior, and thus we can consider that network did not impact on the quality of the obtained data. Concerning the benchmarking applications, they can have a different amount of configuration parameter to set, and thus one configuration phase might take longer than another one. This is not a problem since this kind of launching delay is repeated across execution, since the number of parameters to set does not change, the result is that an application will always be scheduled with a fixed extra delay with respect to the one we have chosen for that application.

Moreover, the time required to xSpark to launch its executors is not fixed, and thus computations might start at a slightly different time when comparing executions. This way different executions of the same composite benchmark might have slightly different execution times.

5.6.2 *External Threats*

We assume that the storage layer is never a bottleneck. In a real world scenario instead it could become one. Spark relies on the storage layer, usually HDFS, that should be designed and sized taking into account the workload. In the tested applications, some stages were storage-bounded, in particular when writing and reading data from the storage. It is important to remember that Spark exploits in-memory processing, so only few stages are affected by this kind of bottleneck. Having a different number of simultaneous executors that read or write from the storage can change the QoS of the storage layer, and thus the overall one.

Another possible threats is the use of skewed input data. The resulting effect is having tasks with a significant different duration in the same stage, due to the fact that, for example in a key-based operation, some of the keys might refer to the majority of the values. The presence of skewed data impacts the performances of Spark and the precision of xSpark. Some of Spark's operations, such as `reduce-by-key`, are optimized in terms of partial data aggregation at each partition to reduce the performance impact of data skewness. xSpark has already

been tested with skewed input data, and it is able to obtain a deadline error less than 2% even with skewed data.

6 USE CASES

xSpark was build by assuming the point of view of the cloud provider. The deadlines for each execution could be either set by the cloud provider itself or by the users of cloud. Moreover the needs of the cloud provider on the fulfillment of the deadlines could vary according to its business model, the provided service and type of users. For example one could be interested on minimizing the number of violations while another the overall error. In the following we present some use cases and related experiments to show how choosing a different strategy, presented in Section 3.6, could help the cloud provider to address its need.

In Table 6.1 we have reported the description of the composite benchmarks that have been used to show how the different use cases behave in the following sections.

6.1 CASE 1

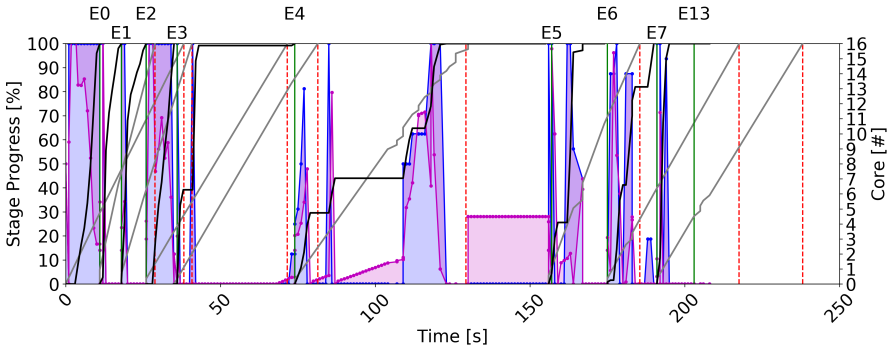
The cloud provider is asked to pay a "fine" which grows with the exceeding computation time required by the application with respect to the previously chosen deadline. We can imagine that the provided deadline is a strict one, which means that results that are available after the deadline have an high chance of being worthless.

In this case, the provider wants to minimize the sum of the errors of the applications that end after the provided deadline. The presence of many delayed applications would reduce the profit of the provider to the point that he could have not accepted some of the applications and still have the same profit.

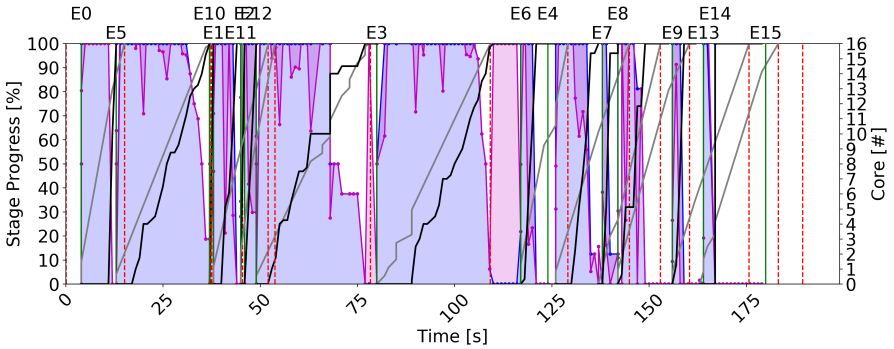
The provider can choose EDF All strategy with $\gamma = 1$, presented in Section 3.6.1, in this way all the resources are allocated to the application whose deadline is the earliest one. We try to reduce the number of applications that will exceed the deadline. With this strategy, only one of the applications will be active at a given time.

The cloud provider needs to pay attention if he decides to use the chosen deadline as a parameter to build up the price of the execution of the application. If a user knows that the current workload is low, he may choose to submit his application with an higher deadline, thus paying less, and still see his application completed as if its deadline was much shorter.

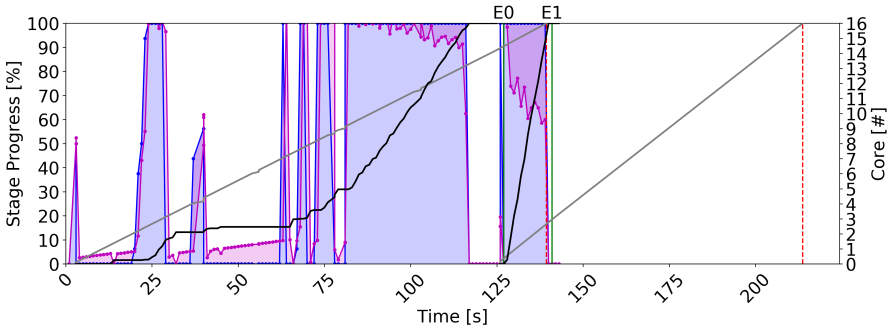
In Figure 6.1 we can see how the different applications of composite benchmark *Mixed 1* behave when using strategy EDF All with $\gamma = 1$. In particular, these diagrams show the allocations of the re-



(a) PageRank



(b) TPCH 21



(c) Aggregate-by-key

Figure 6.1: These diagrams shows the resource allocation of the different applications running in composite benchmark "Mixed 1" with strategy EDF All and $\gamma = 1$. In blue the cores allocated to the executor, in purple the cores requested by the controller. Time axis refers to the beginning of the given application, and not to the beginning of the composite benchmark.

BENCHMARK	APP	DELAY	DL	NR
Mixed 1	PageRank	0	250	671999
	TPCH 21	40	200	26717
	ABK	80	250	319623
Mixed 2	ABK	0	250	319623
	SVM	40	250	46898
	ABK-naive	80	250	486746
Mixed 3	ABK	0	250	319623
	KMeans	40	250	5142187
	ABK-int	80	220	267094

Table 6.1: The composite benchmarks used to discuss the different use cases. From left to right: composite benchmark (*Benchmark*), application (*App*), delay in seconds (*Delay*), deadline in seconds (*DL*), application nominal rate (*NR*). *ABK* stands for *aggregate-by-key*.

sources available on a single worker node to the various applications, in blue the CPU cores allocated to an application meanwhile in purple the ones requested by the controller in order to follow the desired progress rate. It is important to remember that the diagrams should be translated according to their delay that is specified in table 6.1. In general we can see that an application is either active and using all the resources, or inactive with zero CPU cores allocated. This is due to the nature of the used strategy, that aims to allocate all the resources to the currently running application that is closer to its deadline. It is important to understand the deadline of the applications, knowing that *xSpark* is configured to use $\alpha = 0.95$, we can calculate the deadlines of the applications with respect to the beginning of the composite benchmark as $\mathtt{Delay} + \mathtt{Deadline} * \alpha$. In this way, *PageRank* has deadline 237.5 seconds, *TPCH 21* 230 seconds and *aggregate-by-key* 317.5 seconds. Although, this values are only indicative, since different benchmarking tools might have a different time to effectively submit their application. This is not a problem, since the delay error obtained is consistent across different executions of the same composite benchmark. The applications ordered by priority are *TPCH 21*, *PageRank* and *aggregate-by-key*. Differences in priority is visualizable also in the proposed figures. Figure 6.1b represents the allocation of CPU cores to *TPCH 21*, we see that the application is running mostly of the time, since its priority is the highest among the three applications running, when no cores are assigned it means that no tasks are available to be executed by the worker, and thus no resources are allocated. *PageRank* that is plotted in Figure 6.1a completes about the same time *TPCH 21* does, this because even though it has started earlier, it can only use the resources that are not used

BENCH	STRAT	γ	#A	#D	ϵ	ϵ_A	ϵ_D	ϵ_{abs}
Mixed 1	EDF All	1	3	0	23.79	71.37	0	71.17
	EDF Pure	0	3	0	8.59	25.77	0	25.77
	EDF Pure	1	3	0	17.83	53.50	0	53.50
	EDF Prop	0	3	0	6.81	20.43	0	20.43
	EDF Prop	1	3	0	19.41	58.23	0	58.23
Mixed 2	Speed	0	3	0	5.82	17.47	0	17.47
	Speed	1	3	0	15.73	47.20	0	47.20
Mixed 3	Proportional	0	3	0	5.98	17.94	0	17.94
	Proportional	1	3	0	15.23	45.69	0	45.69

Table 6.2: Results of the composite benchmark used to show how the different strategy behave in the proposed use cases. From left to right: composite benchmark (*Bench*), strategy (*Strat*), gamma (γ), number of applications that end in advance (#*A*) and delay (#*D*), average deadline error (ϵ), sum of errors of applications ending in advance (ϵ_A) and delay (ϵ_D), absolute error (ϵ_{abs}).

by the application with highest priority. *Aggregate-by-key* instead, as shown in Figure 6.1c is able to allocate all the resources only after the previous two applications have completed. In Table 6.2 we can see that this strategy is the one with the highest ϵ_A among those strategies tested with composite benchmark *Mixed 1*, since we are trying to maximize the anticipations and reduce the delays.

6.2 CASE 2

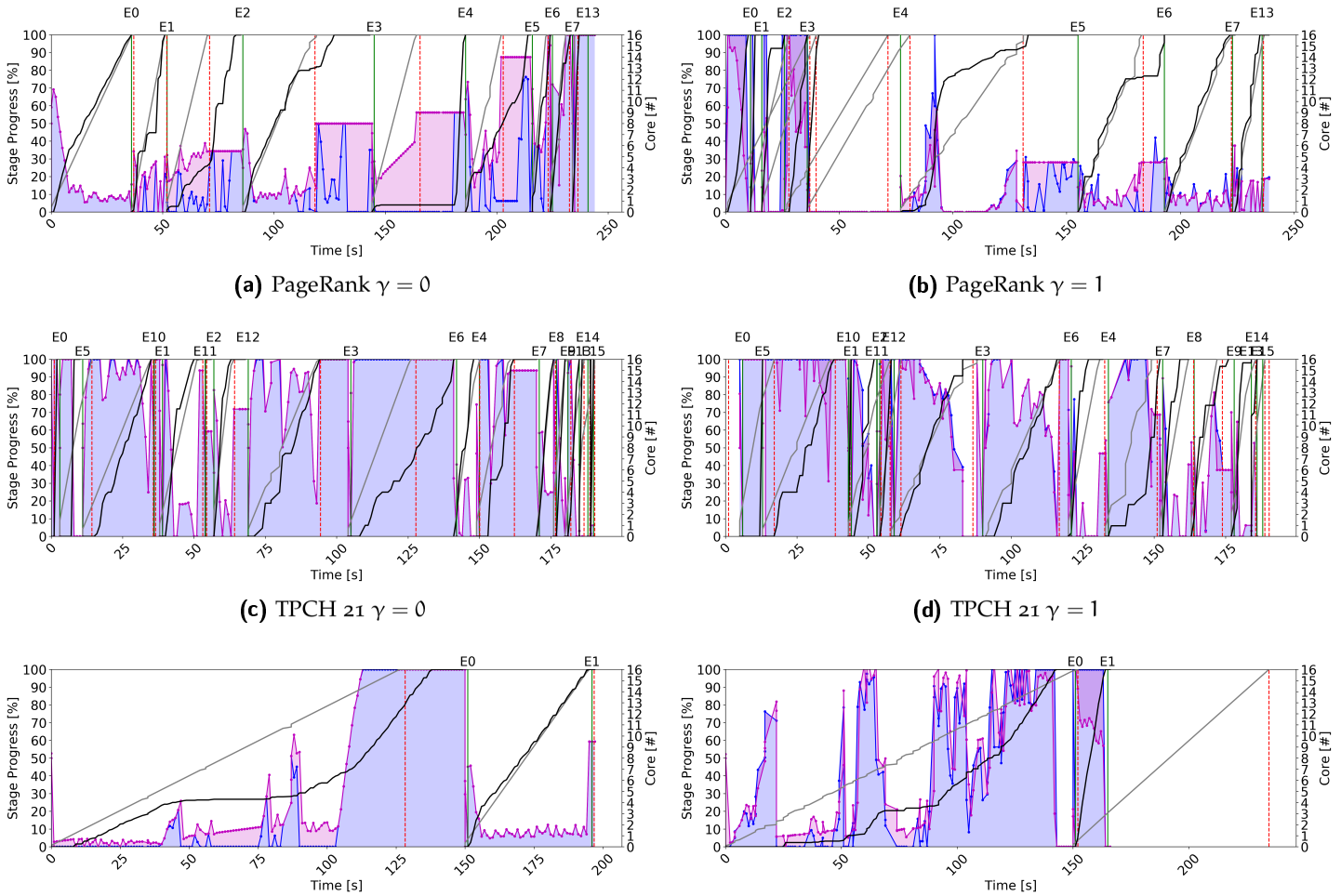
Extending the previous case, the cloud provider wants to be able to use the chosen deadline as a parameter to decide the price of the execution of a given application.

The provider is interested in minimizing also the error of the applications that end before the provided deadline, because if an application ends too early it means that it could have used less resources and still end on time. This means that we want to minimize the absolute error, computed as the sum of the errors of the applications, both those that end early and those late.

The provider can choose *Earliest Deadline First "Pure"* strategy, presented in Section 3.6.2, in this way the application that is closer to its deadline will obtain all the requested resources, possibly leaving free resources for other applications. Each application can allocate resources with a priority that is given by its remaining time to complete.

In this way, different applications can run in parallel, yet there is the possibility that some of the applications are paused.

In Figure 6.2 the cores allocated to each of the applications' executor running on a single worker node are shown, on the left when the



(a) PageRank $\gamma = 0$

(b) PageRank $\gamma = 1$

(c) TPCH 21 $\gamma = 0$

(d) TPCH 21 $\gamma = 1$

(e) Aggregate-by-key $\gamma = 0$

(f) Aggregate-by-key $\gamma = 1$

Figure 6.2: These diagrams shows the resource allocation of the different applications running in composite benchmark "Mixed 1" with strategy EDF Pure and $\gamma \in [0, 1]$. In blue the cores allocated to the executor, in purple the cores requested by the controller. Time axis refers to the beginning of the given application, and not to the beginning of the composite benchmark.

chosen strategy is EDF Pure with $\gamma = 0$ and on the right when with $\gamma = 1$. As in the previous use case example, the applications ordered by priority are *TPCH 21*, *PageRank* and *aggregate-by-key*. Let's begin considering the execution with $\gamma = 0$. From Figure 6.2c it is trivial to understand that *TPCH 21* is the application with highest priority since it acquires all the CPU cores that the controller requests. Same thing happens when *PageRank* and *aggregate-by-key* are the only applications running, the former in Figure 6.2a before the starting of *TPCH 21* (40 seconds from launching), the latter in Figure 6.2e after the ending of other applications (100 seconds from launching). It is important to notice that it is possible that some application has no granted CPU cores, in particular this happens in *aggregate-by-key* that is the application with lowest priority, since all other application have already saturated the available resources. Using $\gamma = 1$ has the effect of distributing all the resources even though controllers are not requesting them all, it is easily visualizable at the beginning of *PageRank* computation in Figure 6.2b and at the ending of *aggregate-by-key* in Figure 6.2f. In Table 6.2 we can compare the results of using the two different γ with EDF Pure, in particular as we can expect, since $\gamma = 0$ is able to satisfy all the deadlines, using $\gamma = 1$ has the effect of increasing the errors, since in general all applications will conclude their execution earlier than in the other case.

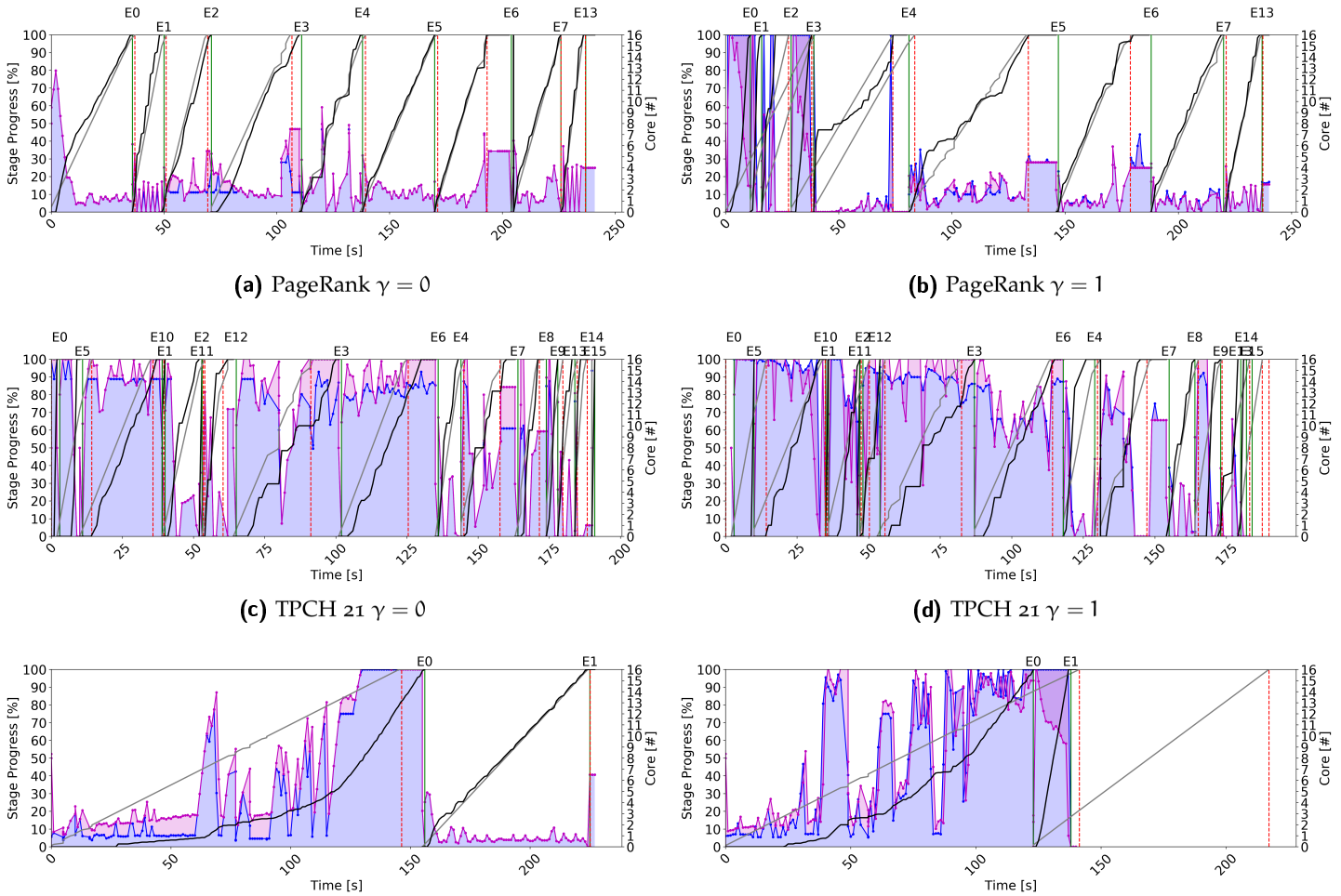
6.3 CASE 3

In the previous cases, there's the risk that an application will never obtain resources until the elapsed time is close to the deadline. This application will suffer of starvation for most of its execution time, since it will acquire resources only when its priority increases. We may want to avoid starvation

In order to work around the problem, we might want to change our allocation strategy to *Earliest Deadline First "Proportional"*, which has been presented in Section 3.6.3. Resource allocation is now proportional to the distance to the deadline, in this way the closest application will still acquire an high amount of the requested resources, yet allowing other applications to acquire some.

The price of choosing this Earliest Deadline First strategy is an increment of the errors in the delayed applications. A provider might be interested in this solution if he wants to execute a large number of applications in parallel and be able to allocate a portion of resources to each of them, while trying to respect the deadlines.

In Figure 6.3 we show the different CPU cores allocation of a single worker node, on the left the result when the composite benchmark is run using strategy EDF Proportional with $\gamma = 0$ and on the right when running with $\gamma = 1$. An application that is running with an higher priority, has more weight and thus can allocate an higher amount



(a) PageRank $\gamma = 0$

(b) PageRank $\gamma = 1$

(c) TPCH 21 $\gamma = 0$

(d) TPCH 21 $\gamma = 1$

(e) Aggregate-by-key $\gamma = 0$

(f) Aggregate-by-key $\gamma = 1$

Figure 6.3: These diagrams shows the resource allocation of the different applications running in composite benchmark "Mixed 1" with strategy EDF Proportional and $\gamma \in [0, 1]$. In blue the cores allocated to the executor, in purple the cores requested by the controller. Time axis refers to the beginning of the given application, and not to the beginning of the composite benchmark.

of CPU cores among those available. In particular, applications have same priority as in the previous use cases, which means that applications sorted by priority are *TPCH 21*, *PageRank* and *aggregate-by-key*. Even though *TPCH 21* has the highest weight in allocating cores, it cannot allocate the entire cluster, this is easily noticeable from the horizontal blue segments in Figure 6.3c about 25 seconds after the beginning of the application, this is due to the fact that the application controller (the purple line) is requesting an higher number of CPU cores than the ones that the application is allowed to acquire given the presence of the other applications. *PageRank* instead has no problem in obtaining the cores its controller has requested, as shown in Figure 6.3a, this happens because in general the amount of cores asked is very low with respect to the ones requested by *TPCH 21*. Last launched application, *aggregate-by-key*, achieves in acquiring all the desired cores only when the other applications have completed, as shown in Figure 6.3e. It is important to notice that in this case, no application is paused as a way to support the execution of other applications. When using $\gamma = 1$ we achieve always complete utilization of the resources, as one can expect. This is noticeable when applications are running alone, for example in Figure 6.3b for *PageRank* and Figure 6.3f for *aggregate-by-key*, respectively the first application to launch and the last to complete in the composite bench. In Table 6.2 we see that using $\gamma = 1$ instead of $\gamma = 0$ has the effect of increasing errors, which is the same thing that happened in the previous example.

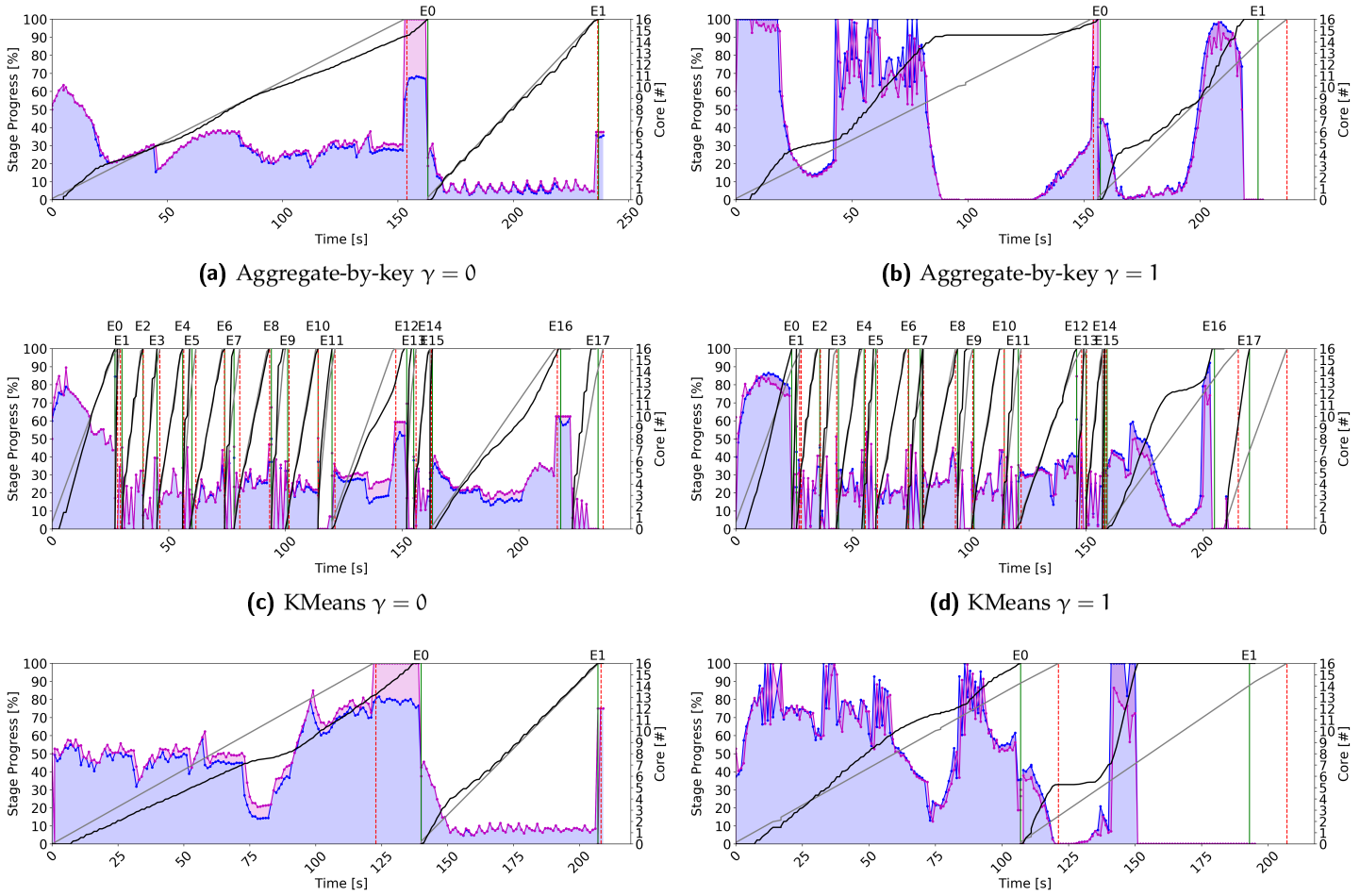
6.4 CASE 4

The cloud provider offers its service to a group of users that belong to the same organization and asks to use the deadline as a parameter to determine the priority of the application. In this way, the deadline is not "strict", which means that it is a non-binding desire on the duration of the execution. The provider will try to respect the deadline, but no (immediate) penalty will be payed.

Since applications are not competing, the provider can decide to use a *Proportional* strategy, presented in Section 3.6.4. In this way, resources will be fairly allocated to the applications according to their requested amount. The resulting allocation will try to follow the desires of each of the applications, without privileging one in particular.

With this approach no application will suffer of starvation and a low average deadline error will be maintained.

In Figure 6.4 we can see the core allocation on a single node when executing the composite benchmark *Mixed 3* with strategy Proportional and $\gamma \in [0, 1]$. From the diagrams of the applications that compose this benchmark, we can see that only in few cases we have resource contention when using $\gamma = 0$, as a result the applications are



(a) Aggregate-by-key $\gamma = 0$

(b) Aggregate-by-key $\gamma = 1$

(c) KMeans $\gamma = 0$

(d) KMeans $\gamma = 1$

(e) Aggregate-by-key-int $\gamma = 0$

(f) Aggregate-by-key-int $\gamma = 1$

Figure 6.4: These diagrams shows the resource allocation of the different applications running in composite benchmark "Mixed 3" with strategy Proportional and $\gamma \in [0, 1]$. In blue the cores allocated to the executor, in purple the cores requested by the controller. Time axis refers to the beginning of the given application, and not to the beginning of the composite benchmark.

able to satisfy their deadline, as shown in Table 6.2. Using $\gamma = 1$ we almost completely remove contention states, by speeding up the execution of applications when not all the available resources would be consumed. As a result, all applications complete their execution earlier, and thus increasing the average deadline error ϵ. In this experiment, using $\gamma = 1$ has not given us any advantage, since all the deadlines were successfully satisfied when using $\gamma = 0$. In Figure 6.4e we can understand how proportional allocation of resources works, indeed we see that the behavior of the blue line (assigned cores) follows the one of the purple one (controller requested cores), this is because the requested cores by the applications build up the weight associated to them. Comparing Figure 6.4a, 6.4c and 6.4e we can understand that in this case contention happens when all applications are running.

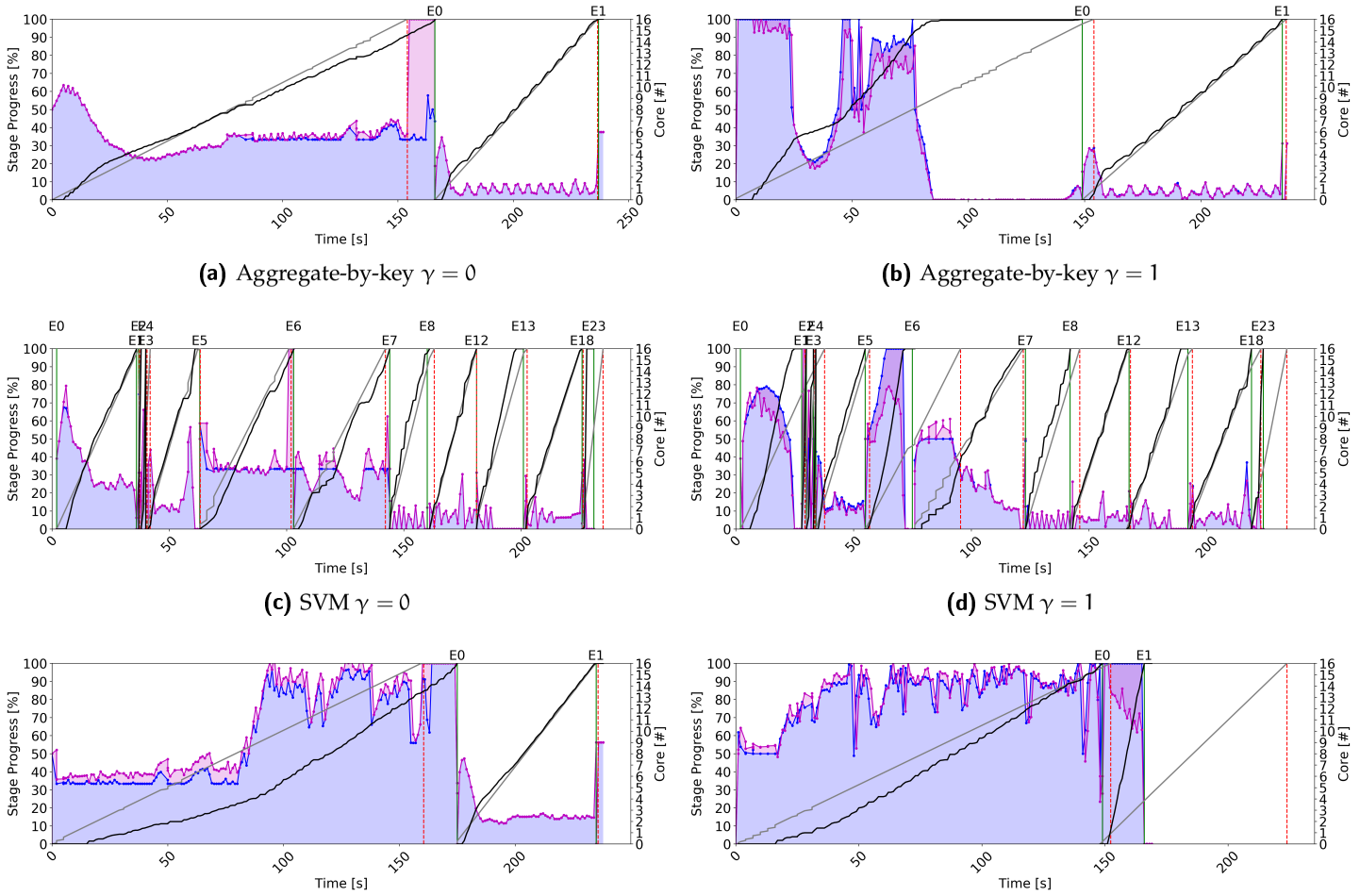
6.5 CASE 5

Extending the previous case, what if applications are heterogeneous and they have a different speed intended as average number of computed records per second (nominal rate)?

The provider might want to choose a distribution strategy that is not proportional to the requested resources, but proportional to the *Speed* of the application, as introduced in Section 3.6.5. In this way we try to favor the applications that are slower by granting more resources.

Choosing this strategy will not result in decrementing the errors in the general case, but will help those applications that are on average slower.

In Figure 6.5 are shown the cores allocated to the various executor running on a worker node, running applications from composite bench *Mixed 2* with strategy *Speed* and $\gamma \in [0, 1]$. From Table 6.2 we see that with $\gamma = 0$ we are able to satisfy all the deadlines with a good average deadline error ϵ, switching to $\gamma = 1$ we decrease the execution time of the applications and thus we increase ϵ_A (sum of deadline error of the applications that end in advance) and ϵ. From the graphs of the applications run using $\gamma = 0$, that are Figure 6.5a (*aggregate-by-key*), 6.5c (*SVM*) and 6.5e (*aggregate-by-key-naive*), we can see how the nominal rate of the applications (*NR* in Table 6.1) is used as a weight to limit the resources assigned to the applications, this limit is plotted as blue horizontal segments, meaning that the system resources were saturated and thus we were not able to allocate more resources to the application than the ones determined by the application weight. Switching to $\gamma = 1$ has the effect of anticipating the completion time of the last application, as shown in Figure 6.5f, this is due to the fact that it is able to allocate the resources of the entire cluster during its last stage, when it is running alone in the cluster.



(a) Aggregate-by-key $\gamma = 0$

(b) Aggregate-by-key $\gamma = 1$

(c) SVM $\gamma = 0$

(d) SVM $\gamma = 1$

(e) Aggregate-by-key-naive $\gamma = 0$

(f) Aggregate-by-key-naive $\gamma = 1$

Figure 6.5: These diagrams shows the resource allocation of the different applications running in composite benchmark "Mixed 2" with strategy Speed and $\gamma \in [0, 1]$. In blue the cores allocated to the executor, in purple the cores requested by the controller. Time axis refers to the beginning of the given application, and not to the beginning of the composite benchmark.

7 CONCLUSION

In the previous sections we have presented the work done in order to support multiple applications in xSpark.

In order to solve the scheduling problem across applications, we have extended the executor controller model. The solution is still proposing a distributed controller, with a centralized heuristic based control loop and a distributed per executor local control loop based on control theory. In order to take into account the presence of multiple applications running together, a per worker node supervisor of the executor controllers has been introduced, in this way we prevent the arising of a resource contention state.

Various strategies have been proposed in order to solve resource contention, which take into account the deadline of an application, i.e. EDF "All", EDF "Pure" and EDF "Proportional", or the cores requested by the executor controller, i.e. Proportional, and the speed of an application in term of its nominal rate, i.e. Speed. All these strategies have been presented and discussed in Section 3.6.

Another important contribution is given in terms of memory allocation. Spark currently does not support the resizing of the memory allocated to an executor, neither for the heap part of the executor process nor for the off-heap region. Since resizing the heap of a Java process is still unfeasible with the latest Java version, we extended xSpark's memory manager in order to support scaling the size of off-heap memory region that an executor can use. In this way, when enabling off-heap allocation, we are able to dynamically resize the off-heap memory region of an executor when applications enter or leave the system, thus we are able to avoid wasting memory. This has been introduced and discussed in Section 3.4.

Using the Clairvoyance EDF analysis, we can understand if a composite benchmark is considered feasible in the sense that an Earliest Deadline First task scheduling is able to satisfy all the applications deadline. When this happens, xSpark is able to complete all the applications in the composite benchmark within the specified deadline, in particular the average error is always around 5% when we are not using EDF "All" strategy. This is a good result considering that we are running xSpark using the parameter $\alpha = 0.95$, which shrinks the deadlines of 5% in order to be more conservative.

When a composite benchmark is not feasible instead, we are able to reduce the overall delay error by keeping the system utilization always at maximum setting the parameter $\gamma = 1$, in particular we can use EDF "All" strategy to reduce the number of violations.

When using strategy EDF "All", xSpark is able to complete composite benchmarks in a time that is comparable with the one obtained from running the applications in vanilla Spark, either serialized or parallelized using static partitioning of resources. Moreover, if provided deadlines are feasible, xSpark is able at the same time of satisfying them, which might not always be true when using vanilla Spark.

In conclusion, we observed that using dynamic off-heap memory allocation has both advantages and disadvantages. When enabled, we are able to achieve a complete utilization of the cluster memory resources, at the price of slightly higher execution times.

Future works

Future works will further investigate resource allocation strategies. Taking into account a penalty cost function allows the controller supervisor to be able to distinguish between strict deadlines, i.e., infinite penalty upon violation, and soft deadlines, i.e., penalty that is proportional to the violation. This way, the supervisor is able to determine which application execution should be favored, in terms of revenue.

BIBLIOGRAPHY

[1] *Apache Hadoop*. URL: http://hadoop.apache.org/.

[2] *Apache Spark*. URL: http://spark.apache.org/.

[3] *Apache Tez*. URL: tez.apache.org.

[4] Chien Hung Chen, Jenn Wei Lin, and Sy Yen Kuo. "Deadline-Constrained MapReduce Scheduling Based on Graph Modelling." In: *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE. 2014, pp. 416–423.

[5] World Wide Web Consortium et al. *Internet Live Stats*. URL: http://www.internetlivestats.com/.

[6] Transaction Processing Performance Council. "TPC-H benchmark specification." In: *Published at http://www. tcp. org/hspec. html* 21 (2008), pp. 592–603.

[7] Michael L. Dertouzos and Aloysius K. Mok. "Multiprocessor online scheduling of hard-real-time tasks." In: *IEEE Transactions on software engineering* 15.12 (1989), pp. 1497–1506.

[8] *Docker Documentation*. URL: https://docs.docker.com/engine/docker-overview/#the-underlying-technology.

[9] Nathan Wayne Fisher. *The multiprocessor real-time scheduling of general task systems*. The University of North Carolina at Chapel Hill, 2007.

[10] Ilya Ganelin, Kai Sasaki, Ema Orhian, and Brennon York. *Spark: Big Data Cluster Computing in Production*. John Wiley & Sons, 2016.

[11] Giovanni Paolo Gibilisco, Min Li, Li Zhang, and Danilo Ardagna. "Stage aware performance modeling of DAG based in memory analytic platforms." In: *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE. 2016, pp. 188–195.

[12] A. Grishchenko. *Spark Memory Management*. URL: https://0x0fff.com/spark-memory-management/.

[13] *Hadoop 1.x Documentation*. URL: https://hadoop.apache.org/docs/r1.2.1/.

[14] *HDFS Architecture Guide*. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[15] Chen He, Ying Lu, and David Swanson. "Real-time scheduling in mapreduce clusters." In: *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE. 2013, pp. 1536–1544.

[16] Kwang Soo Hong and JY-T Leung. "On-line scheduling of real-time tasks." In: *Real-Time Systems Symposium, 1988., Proceedings.* IEEE. 1988, pp. 244–250.

[17] *Java 8*. URL: http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html.

[18] *Java Platform, Standard Edition Tools Reference*. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html.

[19] Kamal Kc and Kemafor Anyanwu. "Scheduling hadoop jobs to meet deadlines." In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. 2010, pp. 388–392.

[20] N. Kozłowski. *Spark Memory Management Part 1 – Push It to the Limits*. URL: https://www.pgs-soft.com/spark-memory-management-part-1-push-it-to-the-limits/.

[21] Palden Lama and Xiaobo Zhou. "Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud." In: *Proceedings of the 9th international conference on Autonomic computing*. ACM. 2012, pp. 63–72.

[22] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. "SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark." In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. CF '15. Ischia, Italy: ACM, 2015, 53:1–53:8. ISBN: 978-1-4503-3358-0. URL: http://doi.acm.org/10.1145/2742854.2747283.

[23] *Micorosft Azure Virtual Machines*. URL: https://azure.microsoft.com/en-us/services/virtual-machines/.

[24] *Microsoft Azure Blob Storage*. URL: https://azure.microsoft.com/en-us/services/storage/blobs/.

[25] Arezou Mohammadi and Selim G Akl. "Scheduling algorithms for real-time systems." In: *School of Computing Queens University, Tech. Rep* (2005).

[26] Andrew Or. *Understanding your Apache Spark Application Through Visualization*. URL: https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html.

[27] Linh TX Phan, Zhuoyao Zhang, Qi Zheng, Boon Thau Loo, and Insup Lee. "An empirical analysis of scheduling techniques for real-time cloud-based data processing." In: *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*. IEEE. 2011, pp. 1–8.

[28] Jorda Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. "Performance-driven task co-scheduling for mapreduce environments." In: *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE. 2010, pp. 373–380.

[29] Poornima Purohit, DR Apoorva, PV Lathashree, et al. "Big Data in Cloud Computing." In: *International Journal of Advance Research, Ideas and Innovations in Technology* 3.3 (2017), pp. 1312–1318.

[30] Jorge L Reyes-Ortiz, Luca Oneto, and Davide Anguita. "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf." In: *Procedia Computer Science* 53 (2015), pp. 121–130.

[31] *Running Spark on Mesos*. URL: https://spark.apache.org/docs/latest/running-on-mesos.html.

[32] *Running Spark on YARN*. URL: https://spark.apache.org/docs/latest/running-on-yarn.html.

[33] *Spark configuration*. URL: https://spark.apache.org/docs/2.0.2/configuration.html.

[34] John A Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C Buttazzo. *Deadline scheduling for real-time systems: EDF and related algorithms*. Vol. 460. Springer Science & Business Media, 2012.

[35] Fei Teng, Frédéric Magoulès, Lei Yu, and Tianrui Li. "A novel real-time scheduling algorithm and performance analysis of a MapReduce-based cloud." In: *The Journal of Supercomputing* 69.2 (2014), pp. 739–765.

[36] *Ubuntu 14.04.5 LTS (Trusty Tahr)*. URL: http://releases.ubuntu.com/14.04/.

[37] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator." In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. URL: http://doi.acm.org/10.1145/2523616.2523633.

[38] Abhishek Verma, Ludmila Cherkasova, Vijay S Kumar, and Roy H Campbell. "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle." In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE. 2012, pp. 900–905.

[39] *Virtual machine sizes for Azure Cloud services*. URL: https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-sizes-specs\#dv2-series.

[40] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, et al. "Flex: A slot allocation scheduling optimizer for mapreduce workloads." In: *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*. Springer-Verlag. 2010, pp. 1–20.

[41] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling." In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 265–278.

[42] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. "Apache Spark: A unified engine for big data processing." In: *Communications of the ACM* 59.11 (2016), pp. 56–65.