Big-Data Streaming Applications Scheduling Based on Staged Multi-armed Bandits

Karim Kanoun, Cem Tekin, *Member, IEEE,* David Atienza, *Fellow, IEEE,* and Mihaela van der Schaar, *Fellow, IEEE*

Abstract—Several techniques have been recently proposed to adapt Big-Data streaming applications to existing many core platforms. Among these techniques, online reinforcement learning methods have been proposed that learn how to adapt at run-time the throughput and resources allocated to the various streaming tasks depending on dynamically changing data stream characteristics and the desired applications performance (e.g., accuracy). However, most of state-of-the-art techniques consider only one single stream input in its application model input and assume that the system knows the amount of resources to allocate to each task to achieve a desired performance. To address these limitations, in this paper we propose a new systematic and efficient methodology and associated algorithms for online learning and energy-efficient scheduling of Big-Data streaming applications with multiple streams on many core systems with resource constraints. We formalize the problem of multi-stream scheduling as a staged decision problem in which the performance obtained for various resource allocations is unknown. The proposed scheduling methodology uses a novel class of online adaptive learning techniques which we refer to as staged multi-armed bandits (S-MAB). Our scheduler is able to learn online which processing method to assign to each stream and how to allocate its resources over time in order to maximize the performance on the fly, at run-time, without having access to any offline information. The proposed scheduler, applied on a face detection streaming application and without using any offline information, is able to achieve similar performance compared to an optimal semi-online solution that has full knowledge of the input stream where the differences in throughput, observed quality, resource usage and energy efficiency are less than 1%, 0.3%, 0.2% and 4% respectively.

Index Terms—Scheduling, Machine learning, Many-core platforms, Data mining, Big-Data, Multiple streams processing, Concept drift.

1 Introduction

Big-Data streaming applications are now widely used in several domains such as social media analysis, financial analysis, video annotation, surveillance and medical services. These applications are characterized with stringent delay constraints, increasing parallel computation requirement and a highly variable stochastic input data stream which have direct impact on the application complexity and the final Quality of Service QoS (e.g., throughput and output quality) [12]. For instance, stream mining applications [1], one of the main emerging Big-Data stream computing applications, are used to classify a high input of variable data stream and are in general modeled using a chain of stages of classifiers and features-extraction tasks (e.g., Figure 1). Different types of dynamically changing data are collected from various heterogeneous sources and multiple types of classifiers are applied on these data to uncover hidden patterns or extract knowledge required for prediction and actionable intelligence applications. In order to adapt to the heterogeneous nature of the data, each stage may integrate different type of classifiers or quality levels and a selection of the processing method is realized at run-time with respect to the predicted type of data. Figure 1 illustrates an example of facial detection application using this application model. The complexity of each task in each stage of the chain may change at run-time with respect to the type of processed input data which is unknown by the application.

Numerous hardware and software solutions have been proposed in order to cope with the increasing complexity and computation requirement of modern streaming applications. At the hardware layer, several many core architectures [9], [20], [21], [22] have been developed to increase the parallelization level and to support the streaming application model. At the software layer, approaches based on load-shedding techniques have been proposed to reduce the workload by selecting the percentage of data that will be processed while other approaches control the processing method of the data streams to adapt to the given allocated resources. However, the majority of state-of-the-art solutions do not handle multiple streams at the same time. Moreover, even in the single stream case, without the support of a proper online smart scheduler that knows how to

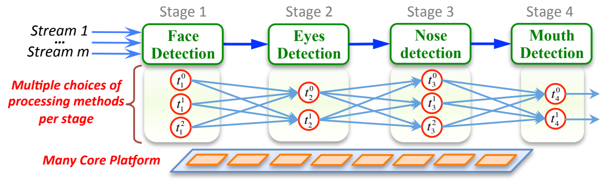


Fig. 1. A stream mining application with multiple tasks per stage [8].

. This work has been partially supported by the YINS RTD project (no. 20NA21 150939), funded by Nano-Tera.ch with Swiss Confederation Financing and scientifically evaluated by SNSF, and the EC H2020 MANGO project (Agreement No. 671668). This research was also supported by NSF grant ECCS-1407712.

Karim Kanoun and David Atienza are with the Embedded Systems Laboratory, Ecole Polytechnique Federale de Lausanne, Lausanne 1015, Switzerland (e-mail: karim.kanoun@epfl.ch; david.atienza@epfl.ch). Cem Tekin is with the Department of Electrical and Electronics Engineering, Bilkent University, Ankara, 06800, Turkey (e-mail:cemtekin@ee.bilkent.edu.tr). Mihaela van der Schaar is with the Department of Electrical Engineering, University of California at Los Angeles, Los Angeles, CA 90095-1594 USA (e-mail: mihaela@ee.ucla.edu).

efficiently coordinate these software optimizations with the real capacity of existing hardware solutions and the dynamically changing application needs, these many core platforms are not able to efficiently handle real time requirements and characteristics of Big-Data streaming application which are dynamically changing at run-time. In fact, existing online scheduling approaches have very limited considerations to the dynamic characteristics of the data streams, which may experience *concept drift* [11] and thus require continuous adaptation. Approaches that rely on offline information are not able to adapt to these concept drifts online.

Finally, energy consumption in many core architectures is becoming a major concern as the cost of powering these type of platforms is significantly increasing [13]. Software techniques presented in the previous paragraph adapt the complexity of stream mining applications at run-time. However, such workload reduction solutions are usually implemented at the application layer which is often oblivious to the system architecture, available system resources or available power management features. Therefore, by combining these software techniques with energy saving features such as *Dynamic Power Management* (DPM) to switch on and off cores, the energy consumption can be reduced without having an impact on the quality of service of the application. In fact, by allocating the proper amount of resources to each task, only required cores are activated. Moreover, the slack time between different application stages can be exploited with DPM when it is detected. Therefore, it is essential that the operating system layer combines techniques from both application and hardware layers in order to maximize the QoS while minimizing the energy consumption.

To address these challenges, we propose a new systematic and efficient methodology and associated algorithms for online learning and energy-efficient scheduling of Big-Data streaming applications with multiple streams on many core systems with resources constraints. The key contributions of this work are as follows:

• We formalize the problem of multi-streams scheduling as a staged decision problem in which the performance obtained for various resource allocations is unknown a priori but learned over time.

• The proposed scheduling methodology uses a novel class of online adaptive learning techniques which we refer to as staged multi-armed bandits. Our scheduler is able to learn online which processing method to assign to each stream and how to allocate resources over time in order to maximize the performance on the fly, at run-time, without having access to any offline information.

• Unlike standard multi-armed bandit problem formulation where each outcome depends only on the latest previous scheduling decision, in our formulation the outcome of each scheduling action depends on a sequence of previous scheduling decisions and feedbacks that are taken at a certain stage (window) of time.

•The regret (i.e., the difference in performance compared to a scheduler that acts optimally from the beginning) of the proposed algorithm increases only logarithmically in the number of rounds.

The proposed scheduler, applied on a multi-stage face detection streaming application in a dynamically changing environment and without using any offline information, is able to achieve similar performance compared to an optimal semi-online solution that has full knowledge of the input stream where the differences in throughput, observed quality, resource usage and energy efficiency are less than 1%, 0.3%, 0.2% and 4% respectively. We also compare our results to a scheduling solution [19] with online learning and concept drift detection. Our scheduler significantly outperforms the solution proposed in [19] in terms of observed quality, obtained throughput, memory usage and complexity.

The remainder of this paper is organized as follows. In Section 2, we describe related work and the benefits of Staged Multi-Armed Bandits. In Section 3, we model our environment including the application, the Big-Data and the platform models. In Section 4, we describe our novel class of online adaptive learning techniques (i.e., our scheduler). In Section 6, we present our experimental results. Finally, we summarize the main conclusions in Section 7.

2 Motivation

2.1 Related Work

Our approach targets a specific type of applications where the QoS depends on both the throughput and the quality observed for each task in the application with a dynamic Big-Data stream under constrained resources. Therefore, we only discuss techniques that have been proposed to adapt Big-Data streaming applications to resource constraints.

The first set of approaches relies on load-shedding [6] [7], where designed algorithms determine when, where, what, and how much data to discard given the desired QoS requirements and the available resources. In [6], the impact of load shedding is known a-priori and the load shedder was decoupled from the scheduler assuming that an external scheduler will handle the assignment of freed resources. In [7], a load shedding scheme ensures that dropped load has minimal impact on the benefits of mining and dynamically learns a Markov model to predict feature values of unseen data. Instead of deciding on what fraction of the data to process, as in load shedding, the second set of approaches [5] [2] [3] [4] [1] determine how the available data should be processed given the underlying resource allocation. In these works, individual tasks operate at a different performance level given the resources allocated to them. They assume a fixed model complexity for each classifier and the variation of the output quality is known a-priori. The problem was formulated as a network optimization problem and solved with sequential quadratic programming. These solutions assume stationary environment while, in reality, data streams are dynamic. Therefore, they may experience a concept drift that requires a continuous online adaptation of the amount of allocated resources to each task and the output quality to maximize the QoS, especially when the resources are constrained. In [11], a survey has been published recently, which categorizes most of the existing concept drift approaches. None of these approaches have been proposed for scheduling Big-Data streaming applications and resource management problems. Recently, in [19], the authors model the scheduling problem as a Stochastic Shortest Path problem and propose a reinforcement learning algorithm to learn the environment dynamics to solve this problem even in the presence of concept drift. However the allocation of the computing resources to each streaming task was not realized by the algorithm. Instead, it assumes that the system knows the amount of resources to allocate to each task to achieve the desired throughput. Moreover, they do not provide a systematic way for the task selection. In section 6.2.4, we compare our results to the scheduler proposed in [19].

To summarize, the above two set of solutions are usually implemented at the application layer and are agnostic to the system constraints and capabilities. Instead, our online learning solution is implemented at the operating system level and it is responsible for resources allocation and processing method selection for each available stream.

2.2 Benefits of Staged Multi-Armed Bandits

In this paper, we model the multi-stream scheduling problem as an online learning problem. Many online learning problems can be formalized using multi-armed bandits (MABs) [15] [14] [17] and efficient algorithms with provable performance guarantees can be developed for these problems. A common assumption in all these problems is that each decision step involves taking a single action after which the reward is observed. Unlike these problems in multi-task scheduling, each decision step (amount of resource to allocate, quality level, etc.) involves taking multiple actions in series corresponding to different types of processing applied on a single data stream and multiple actions in parallel corresponding to different data streams at each stage.

Another class of MAB problems in which the reward at a particular stage depends on the sequence of actions that are taken are the C-MAB problems [16]. However, in these problems it is assumed that (i) all the actions in the sequence are selected simultaneously; hence, no feedback is available between the actions, (ii) the *global* reward function has a special *additive* form which is equal to a weighted sum of the *individual* rewards of the selected actions. Other MAB problems which involve large action sets are [18] where at each time step the learner chooses an action in a metric space and obtains a reward that is a function of the chosen action. Again, no intermediate feedback about the chosen sequence of actions is available before the reward is revealed.

MABs are also used in solving decentralized sequential decision making problems involving multiple learners [31], [32], [33]. However, unlike multi-stream scheduler in which there is a centralized learner, in these problems there are multiple decentralized learners that act on different data streams. The resources are shared among the learners, hence they should carefully select the actions in order to maximize the total reward. But the settings considered in these works are not applicable to multi-stream scheduling because (i) there are no stages; (ii) they cannot adapt based on intermediate feedbacks provided within each stage; (iii) their complexity grows linearly in the size of the action space which is combinatorial in the multi-stream scheduling application. While our staged bandits approach can be extended to involve decentralized decision making, we leave this tedious task as a future research direction and focus on the novel stage decomposition which allows us to learn fast under large number of data streams and concept drift.

Finally, methods such as Q-learning do not fit well into our multi-stream application model. For instance, in Q-learning the feedback space is fixed, and convergence takes place only asymptotically conditional on the fact that every feedback-action pair is sampled infinitely often. One of the most closely related work in Q-learning [37] derives sample complexity bounds on the performance of two variants of Q-learning, by assuming a general discounted Markov Decision Process (MDP) structure. However, the assumptions on the rewards (and the discount factor) are very different from ours. For instance, in the standard MDP model, the reward depends only on the current state and the current action, and is collected after every taken action. In our work, the reward depends on the past sequence of feedbacks and actions, and is collected only at the end of the round.

One of the most famous variant of the sequential decision making problems is the restless MAB problem [31] [36]. Although logarithmic *strong regret* bounds [36] are proven for the restless MAB problem, algorithms that achieve logarithmic strong regret cannot be computationally efficient [41]. For this reason, we choose to learn a myopic benchmark, which can be computed efficiently, and the regret only depends linearly on the number of stages in a round.

3 System Model

We consider a streaming application with multiple streams from different sources. We use s_k to refer to the kth stream and there are N_{stream} streams in total. Processing of these data streams is carried out in a chain of stages [8] [1].

There are $l_{\max}$ dependent processing stages $i \in \mathcal{G}$ with deadlines d_i, where $\mathcal{G} := \{1, 2, \ldots, l_{\max}\}$. Each stage i is composed of a set of tasks $\mathcal{T}_i$ (processing methods). In order to optimize the processing of the incoming data of each stream on the fly, at each stage i, we have multiple tasks to choose from the set $\mathcal{T}_i$. Each task $t_i^j \in \mathcal{T}_i$ at stage i implements a specific processing method that is optimized for a specific characteristics of data with non-deterministic workload. Let N_{task}^i denote the number of tasks in $\mathcal{T}_i$. In our model, the inputs and outputs of stage i depends on the outputs of stage $i-1$. An illustrative system model showing an application with multiple input streams, stages and tasks for a face detection application is given in Figure 1.

The performance of the processing of s_k in stage i is measured by the output quality q_i^k and the amount of workload w_i^k which depends on t_i^j and the characteristic of s_k. In general the data streams can exhibit Big-Data characteristics such as high velocity and high dimensionality so that it is not possible in general to process all the data on time. Therefore, the amount of data that is processed within its deadline gets to the next task in the chain while the remaining unprocessed data are simply discarded. While the majority of prior work assumes stationary data streams, our model is able to work under concept drift.

Finally, our many core platform model is composed of N_{core} cores with idle power saving states *C-states* feature support [23], [24]. *C-states* are core power states that define the degree to which the processor is "sleeping". $C0$ indicates a normal operation (i.e., full leakage power consumption). All other C-states ($C1$-Cn) describe states where the processor clock is inactive (cannot execute instructions) and different parts of the processor are powered down (i.e., reduced leakage power consumption). Deeper C-states have longer wake-up latencies $X_{switch}^{c_k}$ (the time to transition back to $C0$) but save more power. An efficient use of the C-states may then significantly reduce the energy consumption.

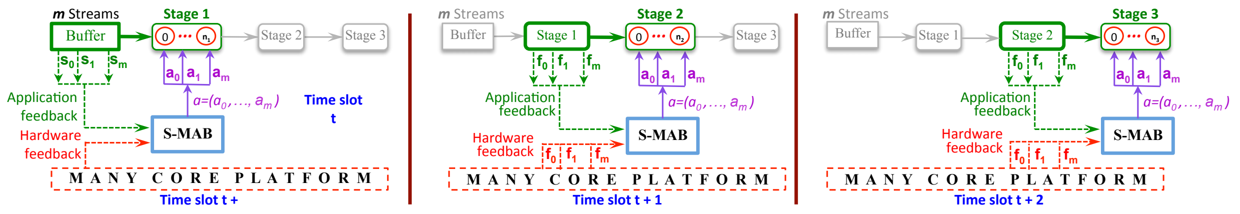


Fig. 2. Example of the execution of the S-MAB during a full round on a 3 stages streaming application

4 A Staged Online Learning Framework

4.1 Problem formulation

Figure 2 illustrates staged processing of the input streams. The system operates in rounds ($\rho = 1, 2, \dots$). At the beginning of each round, multiple data instances arrives from each input stream to the many core platform. The processing of the N_{stream} streams are performed in parallel, as follows. At the beginning of a round, each input stream is assigned to one of the processing methods available in stage 1. After the processing of these data instances are completed or the allowed processing time is consumed, processed data instances of each input stream are assigned to the processing methods of stage 2, and so on. The processing time of an instance at a stage depends on the computing resource allocation, requested quality level and the execution time related to the selected processing task at that stage. We refer to these quantities as actions, and the *joint action vector*[1] at stage i of round ρ is denoted by $\boldsymbol{a}_i^\rho = (a_{i,1}^\rho, \dots, a_{i,N_{\text{stream}}}^\rho)$, where $a_{i,k}^\rho$ represents the action taken for the data of stream k at stage i of round ρ. The set of feasible actions for a data stream at stage i is denoted by $\mathcal{A}_i$.[2] For instance, the total amount of resources allocated to all tasks should be less than or equal to the sum of available resources at each stage.

The number of actions in $\mathcal{A}_i$ is denoted by N_{action}^i. For our application, an action $a \in \mathcal{A}_i$ can be represented as a tuple $a = (t, c)$, where t is the task selected at stage i and c is the amount of resource allocated to that task. Without loss of generality we assume that this holds for the rest of the paper. The set of feasible joint action vectors at stage i is denoted by $\bar{\mathcal{A}}_i := \prod_{k=1}^{N_{\text{stream}}} \mathcal{A}_i$.

After each action $a_{i,k}^\rho$ is taken for data stream k in stage i of round ρ, a feedback $f_{i,k}^\rho$ is observed. Let $\mathcal{F}_i$ be the set of feedbacks that can be observed at stage i at any round for a data stream. We have $\emptyset \in \mathcal{F}_i$. Depending on the stage index (i.e., either the first stage or remaining stages), the feedback observed from each stage i can be composed of one or multiple of these particular elements: (i) Occupancy of the input buffer of each stream k; (ii) The estimated percentage of minimum resources required per task to process a fixed amount of data from stream k; (iii) The selected processing path of the stream k until stage i; (iv) The amount of resources used to process the data of stream k; An explicit definition of the feedbacks for our stream mining application is given in Section 5.2. The joint feedback from all data streams at stage i of round ρ is denoted by $\boldsymbol{f}_i^\rho = (f_{i,1}^\rho, \dots, f_{i,N_{\text{stream}}}^\rho)$. The set of all joint feedbacks at stage i is denoted by $\bar{\mathcal{F}}_i = \prod_{k=1}^{N_{\text{stream}}} \mathcal{F}_i$. In addition, $\boldsymbol{f}_0^\rho$ denotes the joint initial feedback that is available at the beginning of round ρ before any action is taken. The set of all joint initial feedbacks is denoted by $\bar{\mathcal{F}}_0$. For our streaming application, this initial feedback used for the task selection in the first stage is different from other feedbacks used for following stages as it is more related to the status of the buffer rather than a previous stage execution.

Given an action $a_{i,k}^\rho$ for data stream k at stage i of round ρ, let $d_{i,k}^\rho(a_{i,k}^\rho)$ be the random variable which denotes the execution time of task $t \in a_{i,k}^\rho$ for data stream k at round ρ. The deadline of stage i denoted by $d_i > 0$ represents a delay constraint and effects the processing of the data streams in the following way. If $d_{i,k}^\rho(a_{i,k}^\rho) > d_i$, then unprocessed data from the data instance corresponding to data stream k is discarded. Only the processed data gets to the next stage. Therefore, for any round ρ and stage i the set of data streams which do not have any dropped data instance is denoted by $B_i^\rho := \{k : d_{i',k}^\rho(a_{i',k}^\rho) \le d_i, \ \forall \, 1 \le i' < i\}$. Hence, the set B_i^ρ only depends on both the actions and the feedbacks before stage i of round ρ.

Our proposed algorithm select the action in the current stage of the current round based on the feedbacks and actions of past stages in the current round and the feedbacks and actions of the past rounds. In our framework, the joint action to be taken at stage i of round ρ may depend on the set of previously selected actions and observed feedbacks. The set of all sequences of actions is denoted by $\boldsymbol{\mathcal{A}}_{\text{all}} := \prod_{i=1}^{l_{\max}} \bar{\mathcal{A}}_i$. For any sequence of action vectors $\boldsymbol{a} \in \boldsymbol{\mathcal{A}}_{\text{all}}$, let $\boldsymbol{\mathcal{F}}(\boldsymbol{a})$ be the set of sequences of feedbacks that may be observed, and $\boldsymbol{\mathcal{F}}_{\text{all}} := \bigcup_{\boldsymbol{a} \in \boldsymbol{\mathcal{A}}_{\text{all}}} \boldsymbol{\mathcal{F}}(\boldsymbol{a})$. The sequence of actions chosen in round ρ is denoted by $\boldsymbol{a}_{\text{all}}^\rho := (\boldsymbol{a}_1^\rho, \boldsymbol{a}_2^\rho, \dots, \boldsymbol{a}_{l_{\max}}^\rho)$. Let $\boldsymbol{a}^\rho[i] := (\boldsymbol{a}_1^\rho, \dots, \boldsymbol{a}_i^\rho, null, \dots, null)$ represent the sequence of actions chosen in the first i stages of round ρ. Similarly, $\boldsymbol{f}_{\text{all}}^\rho := (\boldsymbol{f}_0^\rho, \boldsymbol{f}_1^\rho, \dots, \boldsymbol{f}_{l_{\max}}^\rho)$ denotes the sequence of all feedbacks observed in round ρ, and $\boldsymbol{f}^\rho[i] := (\boldsymbol{f}_0^\rho, \boldsymbol{f}_1^\rho, \dots, \boldsymbol{f}_i^\rho, null, \dots, null)$ denotes the sequence of feedbacks observed at the first i stages of round ρ. Given a sequence of actions $\boldsymbol{a} \in \boldsymbol{\mathcal{A}}_{\text{all}}$ and sequence of feedbacks $\boldsymbol{f} \in \boldsymbol{\mathcal{F}}(\boldsymbol{a})$ in a round, the reward is drawn from an unknown distribution $F_{\boldsymbol{a},\boldsymbol{f}}$ independently from the other rounds. The expected reward is given by $r_{\boldsymbol{a},\boldsymbol{f}}$. For our Big Data stream mining application, the reward function takes into account the observed quality $q_{i,k}$, the observed throughput $th_{i,k}$ for each stream k in the stage i executing the task in $\mathcal{T}_i$ that is given as an element of $a_{i,k}^\rho$, and finally the amount of unused allocated resources. For our theoretical analysis, we assume that the expected reward is normalized to lie in $[0, 1]$ for all sequences of feedbacks and actions. However, our results will continue to hold (with a constant scaling factor) for any expected reward function that is bounded. An explicit definition of

1. When clear from the context, we will refer to *joint action vector* as the *action*.

2. $\mathcal{A}_i$ also includes the null action, which implies that no action is taken for the stream. The null action is denoted by *null*.

the reward function for our stream mining application is given in Section 5.3.

At stage i of round ρ, the action that is taken for $k \notin B_i^\rho$ is the *null* action (since the instance that belongs to any data stream $k \notin B_i^\rho$ is already discarded in one of the previous stages of that round.) Hence, we only need to select the action for data streams $k \in B_i^\rho$. Let this *constrained action space* at stage i of round ρ be denoted by $\mathcal{A}_i^\rho(B_i^\rho) := \prod_{k \in B_i^\rho} \mathcal{A}_i$. Given the deadline constraint, an algorithm only needs to select actions (tasks selection and allocations) for instances of data streams that are in B_i^ρ.[3]

Every action and feedback sequence is encoded into a *state* by the rule $\phi : \boldsymbol{\mathcal{A}}_{\text{all}} \times \boldsymbol{\mathcal{F}}_{\text{all}} \to \mathcal{X}$, where $\mathcal{X}$ is a finite set. For instance, if $\mathcal{X}$ is taken to be the set of subsets of all data streams, then $\phi(\boldsymbol{a}^\rho[i-1], \boldsymbol{f}^\rho[i-1]) := B_i^\rho$ will denote the set of data streams that do not have any dropped data instance at stage i of round ρ. The probability that feedback $\boldsymbol{f}$ is observed when action $\boldsymbol{a}$ is chosen in stage t when state is x is given by $p_{t,\boldsymbol{a},x}(\boldsymbol{f})$, which is unknown. Since the state is a function of the feedback and action, the state transition probabilities are stage dependent. Due to this, the proposed state model is different than the stationary MDP model assumed in prior works in reinforcement learning [34], [35].

4.2 Myopic benchmark

Since the number of possible sequences of actions and feedbacks that can be taken/observed in a particular round is exponential in $l_{\max}$, it is very inefficient to learn the best sequence of actions by trying each of them separately to estimate $r_{\boldsymbol{a},\boldsymbol{f}}$ for every $\boldsymbol{a} \in \boldsymbol{A}_{\text{all}}$ and $\boldsymbol{f} \in \boldsymbol{\mathcal{F}}(\boldsymbol{a})$. In this section we propose an oracle benchmark called the *Best First* (BF) benchmark whose action selection strategy can be learned quickly by the learner. The pseudocode for the BF benchmark is given in Algorithm 1.

Algorithm 1 Pseudocode for the BF benchmark.

```
1: while $\rho \geq 1$ do
2:     Select action $\boldsymbol{a}_1^{\rho*} = \arg\max_{\boldsymbol{a} \in \mathcal{A}_1^\rho(B_1^\rho)} y_{\boldsymbol{a},\boldsymbol{f}_0^\rho}$
3:     Observe feedback $\boldsymbol{f}_1^{\rho*}$
4:     while $1 < i \leq l_{\max}$ do
5:         $\boldsymbol{a}_i^{\rho*} = \arg\max_{\boldsymbol{a} \in \mathcal{A}_i^\rho(B_i^\rho)} (y_{(\boldsymbol{a}^{\rho*}[i-1],\boldsymbol{a}),\boldsymbol{f}^{\rho*}[i-1]})$
6:         $i = i + 1$
7:     end while
8:     $\rho = \rho + 1$
9: end while
```

Let $\boldsymbol{\mathcal{A}}[i] \subset \boldsymbol{A}_{\text{all}}$ be the set of sequences of actions taken in the first i stages of any round. We will also use $\boldsymbol{f}_{\boldsymbol{a}}[i']$ to denote the sequence of feedbacks to the subset of the actions in $\boldsymbol{a}$ that are taken in the first i' stages of any round. Let $y_{\boldsymbol{a}[i],\boldsymbol{f}_{\boldsymbol{a}[i]}[i-1]} := \mathrm{E}_{\boldsymbol{f}}[r_{\boldsymbol{a}[i],(\boldsymbol{f}_{\boldsymbol{a}[i]}[i-1],\boldsymbol{f})}]$ be the *ex-ante* reward given the sequence of actions $\boldsymbol{a}[i]$ before the feedback for the action vector $\boldsymbol{a}_i$ of stage i is observed, where the expectation is taken with respect to the distribution of the feedback for action vector $\boldsymbol{a}_i$ and state $x = \phi(\boldsymbol{a}[i-1], \boldsymbol{f}[i-1])$.

3. The algorithms we propose in this paper will select the best actions in $\mathcal{A}_i^\rho(B_i^\rho)$ according to an optimality criterion that will be defined later. Since B_i^ρ can be computed using the past sequence of actions and feedbacks, the learner knows that the best action in $\bar{\mathcal{A}}_i$ is always in $\mathcal{A}_i^\rho(B_i^\rho)$. Hence, given the past sequence of actions and feedbacks, taking the action that maximizes the reward over $\mathcal{A}_i^\rho(B_i^\rho)$ is equivalent to taking the action that maximizes the reward over $\bar{\mathcal{A}}_i$.

The BF benchmark incrementally selects the next action based on the sequence of feedbacks observed for the actions of the previous stages. The action that it selects at the initial stage of round ρ is $\boldsymbol{a}_1^{\rho*} = \arg\max_{\boldsymbol{a} \in \mathcal{A}_1^\rho(B_1^\rho)} y_{\boldsymbol{a},\boldsymbol{f}_0^\rho}$.

Let $\boldsymbol{a}_{\text{all}}^{\rho*} = (\boldsymbol{a}_1^{\rho*}, \boldsymbol{a}_2^{\rho*}, \ldots, \boldsymbol{a}_{l_{\max}}^{\rho*})$ be the sequence of actions selected and $\boldsymbol{f}_{\text{all}}^{\rho*} = (\boldsymbol{f}_0^{\rho*}, \boldsymbol{f}_1^{\rho*}, \ldots, \boldsymbol{f}_{l_{\max}}^{\rho*})$ be the sequence of feedbacks observed by the BF benchmark in round ρ. In general $\boldsymbol{a}_i^{\rho*}$, depends on both $\boldsymbol{a}^{\rho*}[i-1]$ and $\boldsymbol{f}^{\rho*}[i-1]$.

At any stage i of round ρ the BF benchmark selects the action in $\boldsymbol{a} \in \mathcal{A}_i^\rho(B_i^\rho)$ that maximizes $y_{(\boldsymbol{a}^{\rho*}[i-1],\boldsymbol{a}),\boldsymbol{f}^{\rho*}[i-1]}$. The total expected *reward* summed over all data streams up to round n by using the BF benchmark is equal to $RW_{\text{BF}}(n) := \sum_{\rho=1}^{n} \mathrm{E}\left[Y_{\boldsymbol{A}^{\rho*},\boldsymbol{F}^{\rho*}}\right]$, where $\boldsymbol{A}^{\rho*}$ is the random variable that represents the sequence of actions selected in round ρ by the BF benchmark, $\boldsymbol{F}^{\rho*}$ is the random variable that represents the sequence of feedbacks observed for the actions selected in round ρ, and $Y_{\boldsymbol{A}^{\rho*}\boldsymbol{F}^{\rho*}}$ is the random variable that represents the reward obtained in round ρ.

Definition of the Regret: Consider any learning algorithm which selects a sequence of actions $\boldsymbol{A}^\rho$ based on the observed sequence of feedbacks $\boldsymbol{F}^\rho$. The regret of this learning algorithm with respect to the BF benchmark in the first n rounds is given by

$$\mathrm{E}[R(n)] := RW_{\text{BF}}(n) - \sum_{\rho=1}^{n} \mathrm{E}\left[Y_{\boldsymbol{A}^\rho,\boldsymbol{F}^\rho}\right] \quad (1)$$

where $Y_{\boldsymbol{A}^\rho,\boldsymbol{F}^\rho}$ is the random variable that represents the reward obtained in round ρ. The regret is defined as the total loss incurred on all data streams up to round n with respect to the BF benchmark. Hence, minimizing the regret implies maximizing the total performance on all data streams. Any algorithm whose regret increases at most sublinearly, i.e., $O(n^\gamma)$, $0 < \gamma < 1$, in the number of rounds will converge in terms of the average reward to the average reward of the BF benchmark as $n \to \infty$. In the next section we will propose an algorithm whose regret increases only logarithmically in the number of rounds.

The definition of regret given in (1) is with respect to the BF benchmark, and hence, is not the strongest notion of regret. Numerous other works such as [36] considered stronger notions of regret, but the algorithms that achieve sublinear strong regret are computationally intractable. Other approaches such as [31] considered weaker notions of regret, in which the regret is computed with respect to the best fixed action. In contrast to these works, the action sequence selected by the BF benchmark depends on the set of observed feedbacks, hence is not fixed. Compared to these two definitions, the use of BF benchmark as the benchmark for regret provides substantial improvements in the learning speed and algorithm complexity. Moreover, there are several important cases in which the BF benchmark is proven to be approximately optimal. For instance, it is shown in [40] that for adaptive submodular reward functions, a simple adaptive greedy policy (which our BF benchmark reduces into under mild assumptions) is $1 - 1/e$ approximately optimal. Hence, any learning algorithm that has sublinear regret with respect to the greedy policy is guaranteed to be approximately optimal. This work is extended to an online setting in [39], where the prior distribution over the states is unknown and only the reward of the chosen sequence of actions is observed. However, an independence assumption is imposed over actions and states to estimate the prior in a fast manner. Using the results in [40], we can show that our BF benchmark is approximately optimal when the reward function is adaptive monotone submodular, an action can only be selected in a single stage and the feedback related with each action is realized at the beginning of each round before action selection takes place. Hence, work on adaptive submodular learning can be viewed as a special case of the S-MAB problem.

5 Feedback Adaptive Learning (FAL): A learning algorithm for the S-MAB problem

In this section, we propose *Feedback Adaptive Learning* (FAL) (pseudocode given in Algorithm 2), which learns the sequence of actions to select based on the observed feedbacks to the previous actions (as shown in Figure 2). FAL learns to select actions in the way that BF benchmark selects actions, hence its regret is measured with respect to BF benchmark.

Let $Y_{\boldsymbol{a}^\rho[i],\boldsymbol{f}^\rho[i]}$ denote the random reward obtained in the first i stages of round ρ. In order to minimize the regret given in (1), FAL balances exploration and exploitation when selecting the actions. Consider the ith stage in round ρ. FAL keeps the following sample mean reward estimates: $\hat{y}_{\boldsymbol{f},i,\boldsymbol{a}}(\rho)$ which is the sample mean estimate of the rewards $Y_{(\boldsymbol{a}^j[i-1],\boldsymbol{a}),(\boldsymbol{f}^j[i-2],\boldsymbol{f},\boldsymbol{f}')}$, $1 \le j < \rho$, $\boldsymbol{f}' \in \bar{\mathcal{F}}_i$ in the first $\rho-1$ rounds corresponding to stage i for which action $\boldsymbol{a}$ is explored after observing feedback $\boldsymbol{f}$ from the action chosen at stage $i-1$. In addition to these, FAL keeps the following counters: $T_{\boldsymbol{f},i,\boldsymbol{a}}(\rho)$ which counts the number of times action $\boldsymbol{a}$ is explored at stage i after feedback $\boldsymbol{f}$ is observed from the action selected at stage $i-1$ in the first $\rho-1$ rounds.

Next, we explain how exploration and exploitation is performed. Let $\boldsymbol{f}$ denote the feedback observed at stage $i-1$, $1 \le i \le l_{\max}$ of round ρ. At the beginning of stage i of round ρ, FAL checks if $\mathcal{U}_i^\rho := \{\boldsymbol{a} \in \mathcal{A}_i^\rho(B_i^\rho) : T_{\boldsymbol{f},i,\boldsymbol{a}}(\rho) < D\log(\rho/\delta)\}$ is non-empty, where $D > 0$ and $\delta > 0$ are constants that are input parameters of FAL whose values will be specified later. If this holds, then FAL explores by randomly selecting an action $\boldsymbol{a} \in \mathcal{U}_i^\rho$ and observes its reward (after observing the feedback $\boldsymbol{f}' \in \bar{\mathcal{F}}_i$) $Y(\rho) := Y_{(\boldsymbol{a}^\rho[i-1],\boldsymbol{a}),(\boldsymbol{f}^\rho[i-1],\boldsymbol{f}')}$, by which it updates $\hat{y}_{\boldsymbol{f},i,\boldsymbol{a}}(\rho+1) = (T_{\boldsymbol{f},i,\boldsymbol{a}}(\rho)\hat{y}_{\boldsymbol{f},i,\boldsymbol{a}}(\rho) + Y(\rho))/(T_{\boldsymbol{f},i,\boldsymbol{a}}(\rho)+1)$. For a round in which FAL explores at stage i, the actions for stage $i+1, ..., l_{\max}$ can be taken arbitrarily or with respect to a predetermined rule (such as the action with the highest reward so far) (cf. Section 5.1). If $\mathcal{U}_i^\rho = \emptyset$, then FAL exploits at stage i by choosing the action that maximizes the estimated reward: $\boldsymbol{a}_i^\rho = \arg\max_{\boldsymbol{a}\in\mathcal{A}_i^\rho(B_i^\rho)} \hat{y}_{\boldsymbol{f},i,\boldsymbol{a}}(\rho)$. Then, the same procedure repeats for the next stage $i+1$.

Setting the parameters of FAL: The number of explorations increases in D (lines 5 and 18), hence setting a larger D results in more accurate reward estimates which leads to better action selections in exploitations. However, this also results in an increase in the reward loss due to explorations. A similar observation can also be made for δ (lines 5 and 18). When δ is small, the probability of choosing a suboptimal action in exploitations is small. However, the number of explorations increases as δ becomes smaller.

Algorithm 2 FAL

```
1: Input $D > 0$, $\delta > 0$, $\boldsymbol{\mathcal{A}}_{\text{all}}$, $\boldsymbol{\mathcal{F}}_{\text{all}}$, $l_{\max}$.
2: Initialize: $\hat{y}_{\boldsymbol{f},i,\boldsymbol{a}} = 0$, $T_{\boldsymbol{f},i,\boldsymbol{a}} = 0$, $\forall \boldsymbol{a} \in \bar{\mathcal{A}}_i, i = 1, \dots, l_{\max}$, $\boldsymbol{f} \in \bar{\mathcal{F}}_i, i = 0, \dots, l_{\max}$. $\boldsymbol{a}_\rho[0] = \emptyset$, $\forall \rho = 1, 2, \dots$
3: while $\rho \ge 1$ do
4:     Find the set of available actions (cf. Section 5.1): $\mathcal{A}_1^\rho(B_1^\rho) = \prod_{k\in B_1^\rho} \mathcal{A}_i$
5:     $\mathcal{U}_1 = \{\boldsymbol{a} \in \mathcal{A}_1^\rho(B_1^\rho) : T_{\boldsymbol{f}_0^\rho,1,\boldsymbol{a}} < D\log(\rho/\delta)\}$
6:     if $\mathcal{U}_1 \ne \emptyset$ then
7:         Select $\boldsymbol{a}_1^\rho$ randomly from $\mathcal{U}_1$, observe $\boldsymbol{f}_1^\rho$
8:         Get reward $Y(\rho) := Y_{\boldsymbol{a}_1^\rho,\boldsymbol{f}^\rho[1]}$
9:         Actions for the remaining stages are selected according to a predefined rule (cf. Section 5.1)
10:        $i^* = 1$, //BREAK
11:    else
12:        Select $\boldsymbol{a}_1^\rho = \arg\max_{\boldsymbol{a}\in\mathcal{A}_1^\rho(B_1^\rho)} \hat{y}_{\boldsymbol{f}_0^\rho,1,\boldsymbol{a}}$ and observe $\boldsymbol{f}_1^\rho$
13:    end if
14:    $i = 2$
15:    while $2 \le i \le l_{\max}$ do
16:        Find the set of streams whose instances are not dropped yet, i.e., $B_i^\rho$
17:        Find the set of available actions (cf. Section 5.1): $\mathcal{A}_i^\rho(B_i^\rho) = \prod_{k\in B_i^\rho} \mathcal{A}_i$
18:        $\mathcal{U}_i = \{\boldsymbol{a} \in \mathcal{A}_i^\rho(B_i^\rho) : T_{\boldsymbol{f}_{i-1}^\rho,i,\boldsymbol{a}} < D\log(\rho/\delta)\}$
19:        if $\mathcal{U}_i \ne \emptyset$ then
20:            Select $\boldsymbol{a}_i^\rho$ randomly from $\mathcal{U}_i$ and observe the feedback $\boldsymbol{f}_i^\rho$
21:            Get reward $Y(\rho) := Y_{\boldsymbol{a}^\rho[i],\boldsymbol{f}^\rho[i]}$
22:            Actions for the remaining stages are selected according to a predefined rule (cf. Section 5.1)
23:            $i^* = i$ //BREAK
24:        else
25:            Select $\boldsymbol{a}_i^\rho = \arg\max_{\boldsymbol{a}\in\mathcal{A}_i^\rho(B_i^\rho)} \hat{y}_{\boldsymbol{f}_{i-1}^\rho,i,\boldsymbol{a}}$ and get the feedback $\boldsymbol{f}_i^\rho$
26:        end if
27:        $i = i + 1$
28:    end while
29:    if Explored (remaining actions are selected according to a predefined rule) then
30:        Update $\hat{y}_{\boldsymbol{f}_{i^*-1}^\rho,i^*,\boldsymbol{a}_{i^*}^\rho}$ using $Y(\rho)$ (sample mean update)
31:        $T_{\boldsymbol{f}_{i^*-1}^\rho,i^*,\boldsymbol{a}_{i^*}^\rho}$ ++
32:    end if
33:    $\rho = \rho + 1$
34: end while
```

The regret of FAL: The regret of FAL can be bounded under two assumptions on the reward structure, which are stated below. The first assumption states that the optimal action is a function of the state $x \in \mathcal{X}$, which is equal to the most recent feedback.

Assumption 1. We have $\phi(\boldsymbol{a}^\rho[i], \boldsymbol{f}^\rho[i]) = \boldsymbol{f}_i^\rho$. For any two length i sequences of action-feedback pairs $(\boldsymbol{a}[i], \boldsymbol{f}[i])$ and $(\boldsymbol{a}'[i], \boldsymbol{f}'[i])$, if $\phi(\boldsymbol{a}[i], \boldsymbol{f}[i]) = \phi(\boldsymbol{a}'[i], \boldsymbol{f}'[i])$, then we have $\arg\max_{\boldsymbol{a}\in\mathcal{A}'_{i+1}} y_{(\boldsymbol{a}[i],\boldsymbol{a}),\boldsymbol{f}[i]} = \arg\max_{\boldsymbol{a}\in\mathcal{A}'_{i+1}} y_{(\boldsymbol{a}'[i],\boldsymbol{a}),\boldsymbol{f}'[i]}$ for any $\mathcal{A}'_{i+1} \subset \bar{\mathcal{A}}_{i+1}$.

The second assumption states that the optimal action for every history of sequence of actions and feedbacks is unique.

Assumption 2. Let $Q_1^*(\mathcal{A}'_1, \boldsymbol{f}_0) := \arg\max_{\boldsymbol{a}\in\mathcal{A}'_1} y_{\boldsymbol{a},\boldsymbol{f}_0}$ and $Q_{i+1}^*(\mathcal{A}'_{i+1}, \boldsymbol{a}[i], \boldsymbol{f}[i]) = \arg\max_{\boldsymbol{a}\in\mathcal{A}'_{i+1}} \{y_{(\boldsymbol{a}[i],\boldsymbol{a}),\boldsymbol{f}[i]}\}$. We assume that $|Q_1^*(\mathcal{A}'_1, \boldsymbol{f}_0)| = 1$ for all $\mathcal{A}'_1 \subset \bar{\mathcal{A}}_1$ and $\boldsymbol{f}_0 \in \bar{\mathcal{F}}_0$, and $|Q_{i+1}^*(\mathcal{A}'_{i+1}, \boldsymbol{a}[i], \boldsymbol{f}[i])| = 1$ for all $\boldsymbol{a}[i] \in \boldsymbol{\mathcal{A}}[i]$, $\boldsymbol{f}[i] \in \boldsymbol{\mathcal{F}}(\boldsymbol{a}[i])$ and $\mathcal{A}'_{i+1} \subset \bar{\mathcal{A}}_{i+1}$, $1 \le i \le l_{\max} - 1$.

The following theorem, whose proof is given in the supplemental material, shows that the regret of FAL with respect to the BF benchmark grows logarithmically in the number of rounds.

Theorem 1. Assume that Assumptions 1 and 2 hold. Let $\Delta_{\min}$ be the minimum of the difference between the expected reward of the best sequence of actions and the second best sequence of actions,[4] where the minimum is taken over all possible feedbacks. When FAL runs with the set of parameters $D = 4/\Delta^2_{\min}$ and $\delta = (2\beta F_{\max} A_{\max} l_{\max} n)^{-1/2}$ we have

$$\begin{aligned}\mathrm{E}[R(n)] \leq 1 + l_{\max} F_{\max} A_{\max} D X \log(2\beta F_{\max} A_{\max} l_{\max}) \\ + 3 l_{\max} F_{\max} A_{\max} D X \log n\end{aligned}$$

where $X = |\mathcal{X}|$, $A_{\max} = \max_{1 \leq i \leq l_{\max}} |\bar{\mathcal{A}}_i|$, $F_{\max} = \max_{0 \leq i \leq l_{\max}} |\bar{\mathcal{F}}_i|$, $\beta = \sum_{t=1}^{\infty} 1/t^2$, and $\mathrm{E}[R(n)]$ is the regret given in (1). Hence, $\lim_{n\to\infty} \mathrm{E}[R(n)]/n = 0$.

5.1 Online management of the action space definition

Each stage of a stream mining application may integrate different type of tasks that differ with their required workload and quality level with respect to the input data in order to adapt to the heterogeneous and dynamic nature of the data (cf. Section 3). To cope with this highly dynamic environment, our FAL algorithm (lines 4, 17) builds its action space on the fly based on the observed feedbacks. The key idea is to have a database that stores all observed feedbacks for each stage and an action space that is built online and customized for each feedback and assigned to it all along the execution. Moreover, these action spaces are retrieved and merged with new actions (if any) generated online each time their corresponding feedbacks are observed again. As demonstrated in Sections 5, the FAL algorithm guarantees a logarithmic increase of the regret in the number of rounds for a defined action space. Therefore, whenever the action spaces of observed feedbacks are stabilized (i.e., when no more new actions are added online), the FAL algorithm executes as expected. Figure 3 and Figure 4 depict the full flow that we apply to generate and maintain a coherent action space all along the execution of the application. In the following, we explain the flow illustrated in these two figures namely, how the exploration mode of the FAL algorithm behaves with respect to the defined action space and how the action space is updated before the execution of each stage.

4. Precise definition of $\Delta_{\min}$ is given in the supplemental material.

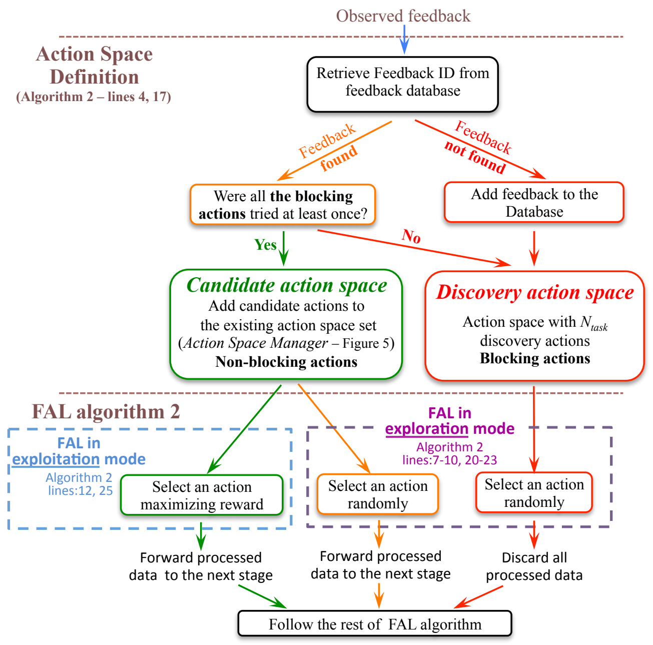

Observed feedback
Action Space Definition
(Algorithm 2 – lines 4, 17)
Retrieve Feedback ID from feedback database
Feedback found
Feedback not found
Were all the blocking actions tried at least once?
Add feedback to the Database
Yes
No
Candidate action space
Add candidate actions to the existing action space set
(Action Space Manager – Figure 5)
Non-blocking actions
Discovery action space
Action space with N_task discovery actions
Blocking actions
FAL algorithm 2
FAL in exploitation mode
Algorithm 2 lines:12, 25
Select an action maximizing reward
FAL in exploration mode
Algorithm 2 lines:7-10, 20-23
Select an action randomly
Select an action randomly
Forward processed data to the next stage
Forward processed data to the next stage
Discard all processed data
Follow the rest of FAL algorithm

Fig. 3. Action space definition: Switching between discovery action space and candidate action space

Each time a new feedback is observed (i.e., not found in the feedback database), the FAL action space switches to discovery mode. As shown in the right part of Figure 3, we initialize the action space related to the newly observed feedback with N_{task} discovery actions. In this actions set, each action $\boldsymbol{a}_i = ((t_i^{j,0}, c_i^{j,0}), .., (t_i^{j,N_{stream}}, c_i^{j,N_{stream}}))$ executes all the streams at stage i with the same task t_i^j (i.e., $t_i^{j,k} = t_i^j$) and the number of cores $c_i^{j,k}$ is equally allocated to each stream k. These actions are mainly used to explore the behavior of each task on each available input data stream for new detected feedbacks. The observed workload $w_i^{j,k}$ and output quality $q_i^{j,k}$ are then recorded in a dedicated structure that keeps track of the observed quality and measured workload of each selected task j for each stream k after each time a stage i is executed. An example of this structure is illustrated in Figure 5. As indicated previously in the FAL Algorithm 1, in exploration mode, actions for the remaining stages are selected according to a predefined rule. In the case where at least one action in the discovery action space remains unexplored, processed data are not forwarded to the next stage. We call these actions blocking actions as the data is discarded immediately after being processed in order to avoid wasting resources on suboptimal tasks selections in the next remaining stages. Once all the blocking actions were tried at least once for the observed feedback, the data structure that holds the measured workload and obtained quality is filled/updated. An action space exploiting these newly recorded data can be safely generated. The FAL action space manager generates then a set of candidate actions based on previous records (cf. next paragraph). Figure 3 illustrates the flow that we use for the selection of action space for each observed feedback. The FAL algorithm enters the exploration mode even with an action space containing generated candidates actions as they were not explored yet. In this case, these actions are non-blocking and the processed data are forwarded to the next stage. This helps minimizing the loss of data and keeps a good overall quality and throughput even in the exploration mode as the generated candidate tasks were already tuned for previous observed feedbacks. In the next paragraph, we explain how we handle the generation of candidate actions.

Our *action space manager*, responsible for the generation of a dedicated action space on the fly for each observed feedback, exploits the quality/workload data structure (Figure 5) built during the exploration mode and which is also continuously updated during the exploitation mode as well. As showed in the flow presented in Figure 4 (steps 1, 2, 3 and 4), we start by finding the r first tasks providing at least the minimum required output quality with the minimum workload for each stream based on previous observations. However, if the minimum output quality is not found then

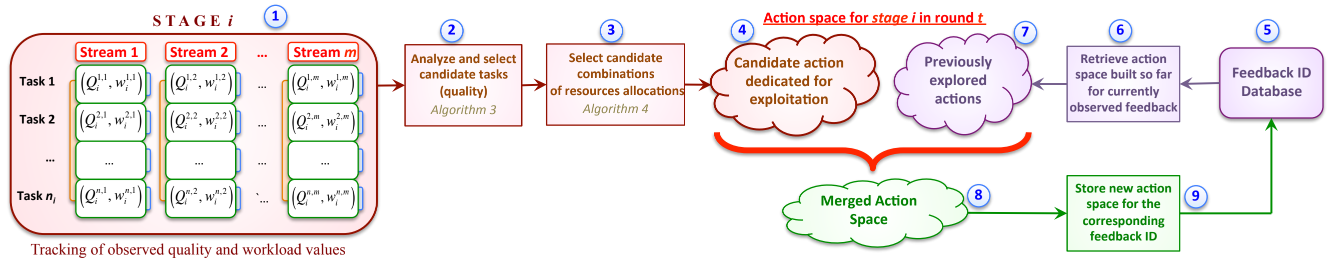

STAGE i
1
Stream 1
Stream 2
Stream m
Task 1
Task 2
Task n_i
Tracking of observed quality and workload values
2
Analyze and select candidate tasks (quality)
Algorithm 3
3
Select candidate combinations of resources allocations
Algorithm 4
Action space for stage i in round t
4
Candidate action dedicated for exploitation
7
Previously explored actions
6
Retrieve action space built so far for currently observed feedback
5
Feedback ID Database
Merged Action Space
8
Store new action space for the corresponding feedback ID
9

Fig. 4. Action space manager: Online update of the action space before the execution of each stage during each round

we select r tasks with the r first maximum output quality. Algorithm 3 illustrates the pseudocode for the candidate tasks selection for $r = 1$. Once candidate tasks of each stream are selected, we select candidate of combinations of resources allocation. Algorithm 4 illustrates a pseudocode of the algorithm that we use to generate candidate cores allocation for the pre-selected candidate tasks. First, we compute the total number of cores required for all the streams (*lines 1-7*). Then, we assign a minimum number of cores for each stream based on the percentage of its workload with respect to the total workload (*lines 7-9*). However, it may happen that the real schedule would require more than the pre-selected cores allocation as the workload of data are different. Moreover, the processing of one single data cannot be divided between the cores. Thus, in *line 10*, we generate all combinations of cores allocations satisfying the previously computed estimated cores allocation plus $1, 2, ..., h$ cores for each stream. Among all generated candidate actions, we discard those with allocations that exceed N_{cores} (*line 11*). Then in steps (5) (6) and (7), we retrieve the action space built in previous rounds (if any) for the observed feedback. Finally, in steps (8) and (9), we merge this action space with the newly generated action space. The action space is now fully updated and ready to be processed by the rest of the FAL algorithm. Unlike discovery actions, the candidate action space guarantees a minimum amount of quality and throughput which allows the processed data to be safely forwarded to the next stage even when the FAL algorithm is in exploration mode. These actions are non-blocking actions.

5.2 Feedback space definition: Exploiting feedbacks for concept drifts detection

In reality, data streams are dynamic. They may then experience a concept drift that requires a continuous online adaptation of the task selections and the amount of allocated resources to each task to maximize the QoS especially when the resources are constrained. Therefore, the observed feedbacks parameters should be selected in a way that they reflect these variations at run-time to the FAL algorithm. To track the characteristics of the buffer of stream k continuously at run-time, we chose two feedback parameters $f^{0,k}_{buff}$ and $f^{1,k}_{buff}$. The first parameter $f^{1,k}_{buff}$ indicates the occupancy of the buffer of each stream k to illustrate the number of data in the buffer. The second parameter $f^{1,k}_{buff}$ indicates the estimated percentage of minimum resources required per task to process a fixed amount of data from stream k. This estimated percentage can be computed using the recorded average workload in previous rounds, a fixed number of data (e.g., half size of the buffer) and the capacity of the core. The first feedback parameter triggers a new feedback when

Algorithm 3 Candidate tasks selection

```
1: Input $w_i, q_i, N_{stream}, N_{task}, q^{min}$.
2: for $k$ in $N_{stream}$ streams do
3:     $selTask[i] = \arg\min_{0 \le j \le N_{task}} (w_i^{j,k} \mid q_i^{j,k} \ge q_i^{min,k})$
4:     if $selTask[i]$ not initialized then
5:         $selTask[i] = \arg\max_{0 \le j \le N_{task}} (q_i^{j,k})$
6:     end if
7: end for
```

Algorithm 4 Generating candidate core allocations

```
1: Input $w_i^{j,k}$, $d_i^{j,k}$, $N_{core}, N_{stream}, h$.
2: for $k$ in $N_{stream}$ streams do
3:     $total\ workload += w_i^{j,k} * d_i^{j,k}$
4:     $\#cores += \lceil \frac{w_i^{j,k} * d_i^{j,k}}{core\ capacity} \rceil$
5: end for
6: $\#cores = min(N_{core}, \#cores)$
7: for $k$ in $N_{stream}$ streams do
8:     $\#cores[i] = \frac{w_i^{j,k} * d_i^{j,k}}{total\ workload} * \#cores$
9: end for
10: Generate all combinations of allocations where each stream $i$
    core allocation action ranges from $\#cores[i]$ to $\#cores[i] + h$
11: Discard actions with allocations that exceed $N_{core}$ cores
```

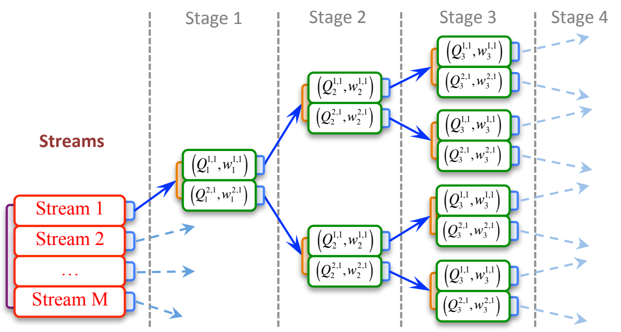


Fig. 5. Example of a quality/workload tracking structure for an application with 2 tasks per stage

the number of data changes while the second parameter triggers a new feedback when the type of data changes. An additional feedback illustrating the observed quality can be also applied as well to detect new observations. Then, the FAL algorithm guarantees that each task in the first stage is tried at least once for the new detected feedback before forwarding the processed data to the next stage.

A concept drift may also appear in one of the stages of the chain. The concept of detecting these variations in these stages is also similar to what we have described previously for the buffers. However, we change the feedback parameters that we observe as these feedbacks are now related to the results of their previous processing stages rather than a buffer status. In fact for a stage i, we observe 2 parameters $f_i^{0,k}$ and $f_i^{1,k}$ (with $0 \le i \le l_{max}$) for each stream k namely, the selected processing path of the stream k until stage i and the amount of resources used to process the data of stream k. For the first parameter $f_i^{0,k}$, the processed path can be computed using the selected task indices in previous stage, this parameters allows the reward value (we discuss *the reward* in the next Section) to be a performance indicator of the different available processing paths. For the second parameter $f_i^{1,k}$, we only take into account the resources that were actively used. In other words, if the scheduler decides to allocate M cores and these cores were active 70% of the duration of the time slot, then the amount of resources used is $0.7 * M$. This parameter allows to trigger new feedbacks when the number of input data from previous stage or the required workload has changed. Moreover, a variation in the workload can be highly implied by a variation in the type of input data. Finally, our feedback parameters are then fully independent from the nature of the application and can be applied on any streaming applications that adopts a chain model with multiple tasks per stage.

5.3 Reward: quality and throughput maximization and energy consumption minimization

The process of selecting the action space on the fly provides an estimated lower bound and upper bound of the total required resources as described in Algorithm 4. Moreover, the size of action space of each feedback is increasing online. Therefore a meaningful metric is required by the FAL algorithm in order to guide the algorithm to choose the right actions among all available actions. This metric is the reward that is attributed for each action taken for each feedback. In other words, when a quality q_i^k and throughput th_i^k are observed for taking action $\boldsymbol{a}_i^\rho$ for feedback $\boldsymbol{f}_{i-1}^\rho$ a reward r_i is assigned for the tuplet $(\boldsymbol{f}_{i-1}^\rho\ ,\ \boldsymbol{a}_i^\rho)$. These reward values are used by the FAL algorithm before an action decision is taken (lines 12 and 25). Our reward function takes into account the observed quality q_i^k, the observed throughput th_i^k for each stream s_k in the stage i executing the action $a_{i,k}^\rho$ and finally the amount of unused allocated resources. We define $r_i = (\sum_{k=0}^{N_{\text{stream}}} q_i^k) << 6\ digits + (\sum_{k=0}^{m} th_i^k) << 3\ digits + u_{cores}$, where u_{cores} represents the number of remaining unallocated cores. Each 3 digits in the final reward value represents an integer reward value related to one of the considered metrics to optimize (i.e., quality, throughput, resource usage). Since we can only have one single integer reward value to represent all the three metrics at a time, we sum the value of the quality (shifted by 6 digits), the throughput (shifted by 3 digits) and the resource usage as showed in Figure 6. Recalling that the primary goal of the FAL algorithm is to maximize the obtained reward, therefore by setting these values in this order, the reward function guides the FAL algorithm to first maximize the quality then the throughput and finally to minimize the amount of unused allocated resources. In fact, two actions with different resources allocations may generate the same quality and throughput, however in this case, a higher reward is assigned to the action that allocate less resources

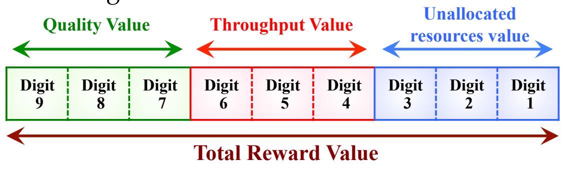


Fig. 6. Illustrating the observed quality, throughput and resource usage in one single reward integer value

(due to the least 3 significant digits of the reward function, i.e., the number of unallocated cores). The leakage energy consumption is then reduced.

6 Experiments

6.1 Experiments setup

We implemented both our S-MAB scheduler agent and environment in C. We also developed a stream mining application for face detection similar to the model presented in Figure 1 and Section 3. We use Haar feature-based cascade classifiers [26] in OpenCV [25]. The developed face detection application is composed of 4 stages. Stage 1 detects the face, stage 2 detects the eyes, stage 3 detects the nose, and stage 4 detects the mouth. Each stage executes a Haar classifier trained to detect its object of interest. Several parameters can be tuned in the Haar classifier in order to control the false detection rate and its computational complexity. Moreover detecting these objects sequentially increases the overall classification accuracy and decreases the required execution time. For instance, when a face is detected in Stage 1, in Stage 2 it is more efficient to look for the eyes only inside the detected face (instead of the full image) reducing then the complexity and false alerts. The same idea/concept can be applied for the remaining stages. In our experiments, we use 4 databases [27] [28] [29] [30]. These databases have different characteristics (e.g., image size, face size...), which impact the workload intensity and the required tuning of the classifier for each stage. For instance, by specifying to the classifier the minimum possible object size for each stage, where objects smaller than that size are ignored, both the output quality and the workload can be controlled. Moreover, there are correlations between the size of the face, the eyes, the nose and the mouth that can be exploited to select the right minimum size for each stage. Thus, we generate multiple configurations of Haar classifiers for each stage and we select a subset of images from each database to use for our experiments. We model then N_{task} tasks of a stage i with N_{task} configurations of Haar classifiers. We choose these configurations such that for each images database, there is at least one path that produces 100% of accurate results and which is unknown by the S-MAB scheduler.

Figure 8 shows the variation of the workload and the observed quality for stage 1 with respect to the selected task and the database serving the image. For instance, DB3 has maximum quality for each possible task in stage 1. However, the workload measured for Task 3 for DB3 is significantly less than the one measured for Task 1. Therefore, the scheduler has to first learn how and when to explore all the tasks (since the data characteristics are dynamically changing over the time) and then to exploit it by choosing the task with the minimum workload and providing the best required quality. Following the same previous explanation, an efficient processing of DB1, DB2, DB3 and DB4 would require their execution with tasks 2, 2, 3 and 1 respectively.

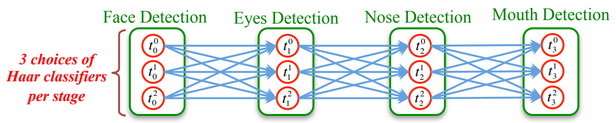


Fig. 7. 81 processing paths in our face processing application with 4 stages and 3 tasks/stage.

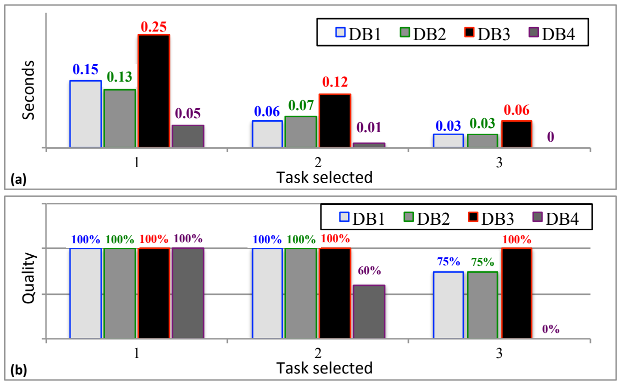


Fig. 8. Stage 1: workload and quality variation with respect to the selected tasks and the image database. (a) Average measured execution time for processing 10 images at stage 1. (b) Average observed quality (i.e., percentage of the detection of the object of interest in 20 images) at stage 1.

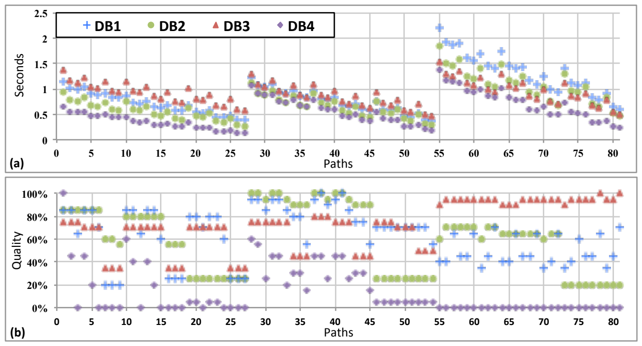
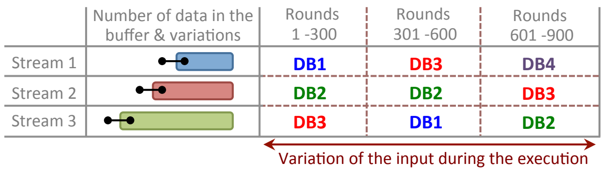


Fig. 9. Workload and quality variation with respect to each full processing path and the image database. (a) Average measured execution time for fully processing 10 images. (b) Minimum average observed quality (i.e., percentage of the detection of the object of interest in 20 images in the stage having the minimum percentage value)

In stream mining applications, the workload and quality of each task in each stage depend on selected tasks in previous stages. Therefore, the selection of the tasks becomes less trivial when the application has several stages. Figure 9 depicts the measured workload and observed quality variation with respect to each full processing path in the case of our face recognition application with 81 possible processing paths (i.e., 4 stages with 3 tasks/stages). Figure 9 (a) shows the measured workload accumulated among the 4 stages while Figure 9 (b) shows the quality (here the quality is the percentage of the detection of the object of interest) for the stage that recorded the minimum quality level during the full processing path. All these informations are unknown by the scheduler and have to be learned online.

Finally, in order to simulate concept drifts, in Figure 10, we show how the input streams of our application are mapped to the 4 different databases all along the execution of 900 rounds. We also specify how big is the buffer of each stream compared to each other. In our experiments, we simulate a platform of 64 homogeneous cores with the



	Number of data in the buffer & variations	Rounds 1 -300	Rounds 301 -600	Rounds 601 -900
Stream 1		DB1	DB3	DB4
Stream 2		DB2	DB2	DB3
Stream 3		DB3	DB1	DB2

Variation of the input during the execution

Fig. 10. Experimental setup: mapping 4 different databases to 3 input data stream for 900 rounds.

same capacity C. For the sake of clarity of the experimental section we fix the deadline of each stage at design time. Stage 1, 3 and 4 are given loose deadlines, while Stage 2 is given a very short deadline in a way that it will not be possible to process all the data at that stage (to simulate the workload intensity of Big-Data). The goal of our experiments is then to show that our S-MAB scheduler is able to find the correlations between the different stages and to exploit its sequential model to find the right processing path for each stream even when the databases change on the fly (i.e., in the presence of concept drift) and to select the right resources allocation strategies that fit the targeted minimum quality and without any prior knowledge on the relation between the processing stages, the used databases characteristics and the buffer sizes.

6.2 Experiments results

In the following, first we explain in details the experimental results obtained for stage 1 namely, exploration and exploitation phases, actions selections, obtained throughput, observed quality and allocated resource usage (Figure 11). Then, we generalize our results for the remaining stages of the application (Figure 12). Another feature of our scheduling algorithm is the possibility for the user to select which minimum processing output quality is required. Therefore, we also provide experimental results with a minimum output quality set to 80% (Figure 13) and we compare the selected resource allocation for each stream with the previous results (Figure 14). Next, we compare our results with an existing Big-Data stream mining applications scheduler [19] that adopts a reinforcement learning technique and integrates a concept drift detection feature (Figure 15 and 16). Finally, we compare our results with a second scheduling solution that has full knowledge of the streams, workload and tasks quality at design time. We then compare the amount of saved leakage energy, the throughput, and observed quality for the face recognition application (Figure 17) and for a set of different configurations of a synthetic application (Figure 18) modeling Big Data stream mining applications.

6.2.1 *Illustrating S-MAB scheduler main features with experimental results observed in Stage 1*

As already discussed in the theory part (Section 5.2), the feedback used for the task selection in the first stage is different from other feedbacks used for following stages as

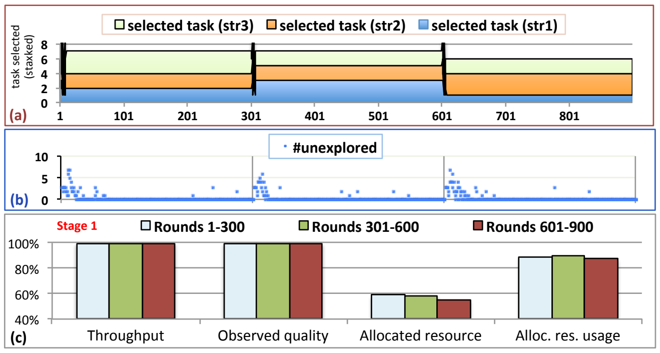


Fig. 11. Stage 1 execution results: (a) Evolution of the task selection for each stream (stacked). (b) Evolution of the number of unexplored actions. (c) Evolution of the obtained throughput, observed quality, allocated resource, and allocated resource usage.

it is more related to the status of the buffer rather than a previous stage execution. Figure 11 (a) shows the selected task index (stacked) at stage 1 for each stream during each round. The figure shows then that each time the scheduler detects a variation in the characteristic of the input stream, the scheduler goes into exploration mode for few rounds. Once all the tasks are (re-)explored for this stage, the scheduler goes back into exploitation mode. This concept is illustrated around rounds 1, 300 and 600 which exactly corresponds to where we have generated concept drifts in our experimental setup as showed before in Figure 10. The idea of switching between exploration mode and exploitation mode with respect to the detected input stream characteristics is also illustrated in Figure 11 (b) which shows the evolution of the number of unexplored actions. In Figure 11 (b), there are two types of unexplored actions. First, unexplored discovery actions, appearing in (b) when new tasks are explored in (a), are the actions that block the data at this stage since not all the tasks were explored yet. Second, the unexplored actions that appear even when the task selection is stabilized in (a), are the candidate actions that are added at run-time to the action space of the already explored feedbacks. These candidate actions only add new resource allocation configuration without changing the selected task. Moreover, these candidate actions do not block the stream (the processed stream is forwarded to the next processing stage) when the FAL algorithm is in exploration mode. Therefore and as showed in Figure 11 (a) and (b), a feedback is considered explored once the number of unexplored discovery actions is 0 (i.e., tasks selection optimized) and fully explored once the number of unexplored discovery and candidate actions is 0 (i.e., resource allocation optimized).

Finally, Figure 11 (c) depicts the evolution of the overall (i.e., all streams accumulated) obtained throughput, observed quality, allocated resources and allocated resources usage with respect to the data variation phases (i.e., with respect to the round index). For Stage 1, the deadline is set in a way that is possible to process all the data in the buffer. Moreover, only 3 processing paths are available in Stage 1. Therefore, the adaptation of the throughput and the quality to the different simulated data variations is straight forward. Figure 11 (c) shows that the throughput and the observed quality were kept over 99% with a usage around 90% of the allocated resources (e.g., for rounds 1-300, allocated resources = 58.9% and used resources = 52.3%). In fact, in the exploitation mode, the scheduler chooses for each stream the task that provides the maximum reward obtained during the exploration phase (i.e., maximum quality with the least amount of workload and minimum resources). For instance from rounds 1 to 300, tasks 1, 1, and 3 are selected for stream 1, 2 and 3 respectively (as shown in Figure 11 (a)) which is also in reality the optimal selection for DB1, DB2 and DB3 respectively for this stage (as shown in Figure 10 (a) and (b)). We validate the optimality of the resources allocation later in Section 6.2.5 by comparing to a scheduling solution that has full knowledge of the streams, workload and tasks quality at design time.

6.2.2 Generalizing the results to the remaining stages

Figures 12 (a), (b) and (c) illustrate the evolution of the throughput, observed quality, allocated resources and allocated resources usage obtained for stage 2, 3 and 4 respectively and using the same experimental setup applied in the previous Section. We only discuss the results that are different from the one obtained in the first stage. In our steam mining application model, the output quality and the workload of each stage depends on selected tasks in its previous stages. Therefore, now there are 9, 27, 81 possible processing paths for stage 2, 3, 4 respectively. In these stages, a feedback is characterized by the path index taken by the data stream and the amount of its resources usage. In stage 2 (i.e., Figure 12 (a)), the deadline is set in a way that it is not possible to process all the data in the buffer, therefore our scheduler reduces the throughput to a value between 60% and 80% depending on the characteristic input stream while the observed quality, allocated resources and resource usage are kept around 100%. After the exploration phase (i.e., around rounds 0, 301 and 601) the scheduler decides to lower the throughput in order to maximize the output quality. The decrease of the overall throughput observed in rounds 301-600 (Figure 12 (a)) is due to the increase in the overall workload of stream 3 when assigned to DB1. In stage 3 and 4, the quality is kept over 99% most of the times while the allocation usage is around 90% which minimizes the waste of leakage energy (experimental results related to energy consumption are presented in Section 6.2.5). Even though the average allocated resource is less than 80% in Stage 3 and Stage 4, the throughput shown in the figures did not reach its maximum value. This is due to the blocking actions that were taken in previous stages for exploration purpose. In fact, when a blocking action is taken, the stream is discarded in the following stage and a throughput value of zero is recorded for the remaining stages in that round. However, when measuring the throughput only for rounds where the streams are not discarded, then we obtain a throughput value over 99%.

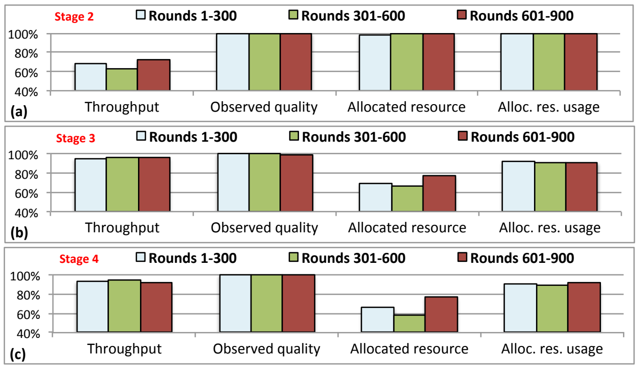


Fig. 12. Evolution of the obtained throughput, observed quality, allocated resource and allocated resource usage in: (a) Stage 2, (b) Stage 3 and (c) Stage 4.

6.2.3 Synchronizing with minimum user quality requirements

In this experiments set, we show how our scheduler can adapt its scheduling decision to the minimum required quality output. For instance, if the user is satisfied with only 80% of the possible maximum quality, the scheduler adapts its scheduling decision in a way that it finds the processing path that gives a quality level between 80% and 100% while providing the maximum possible throughput. In fact, a lower output quality does not imply less workload. For instance, decreasing the minimum possible object size parameter in a Haar feature-based cascade classifier for object detection increases the classifier sensitivity and more importantly its workload. However an increase in the sensitivity may imply an increase in false detections thus providing an output quality less than 100% with a higher workload. Figure 10 illustrates this concept. In fact, the figure shows that for DB3, the maximum quality is obtained for path 81, while other paths provide lower quality but with a higher workload. The scheduler should not then lower the quality of stream mapped to DB3 even if the user allows it as it will decrease the throughput. To validate this feature of our scheduler, we run the same experiment setup as in previous section but with a minimum quality set to 80%. Figure 13 depicts full details of observed quality per stream. The figure shows that in fact DB3 (i.e., stream 3 round [1, 300]; stream 1 round [301, 600]; stream 2 round [601, 900]) was kept at its maximum quality in order to maximize the throughput. DB4 (stream1 round [601, 900]) was also kept at its maximum quality as based on Figure 10, the only quality that can be observed above 80% is 100% (i.e., path 1). Finally, the quality of remaining streams were successfully decreased to between 80% and 90%. Figures 14 compares the resource allocation realized for experiments with minimum quality 100% and 80% respectively. The figure shows that, the new task selection and resource management actions applied by our scheduler allowed to keep the processing quality level of each stream above 80% (as allowed by the user) while keeping the maximum throughput and allocating less resources than the experiment realized in previous Section (i.e., compared to minimum quality of 100%). Finally, in Figure 13 streams that kept an output quality of 100% were allocated the same amount of resources when compared to previous experiments while other streams were assigned with less resource.

6.2.4 Comparison with an existing Big-Data stream mining solution

In this last experiments set, we compare the results of our scheduling approach to a recent Big-Data stream mining scheduling solution [19] from the literature. In [19], the scheduling problem was formalized as a Stochastic Shortest Path (SSP) problem and a reinforcement learning algorithm was proposed to learn the environment dynamics. However, the allocation of the computing resources to each streaming task was not realized by the algorithm. Instead, it assumes that the system knows the amount of resources to allocate to each task to achieve the desired throughput. Moreover, the SSP solution was applied on a single stream and it does not provide a clear and systematic way to chose the tasks of each stage in exploration mode. To adapt [19] to our experimental setup, we have applied the following modifications:

First, for the cores allocation, we allocate the number of minimum required cores given the input size, the desired throughput and the average measured workload of the selected task from previous rounds. Second, to handle the multi-stream model, we assign a dedicated SSP optimizer to each stream. Since there are no priorities among the streams, we divide the resources equally among the streams. For instance if the platform has 60 cores, and 3 stream inputs, then, each SSP will have 20 cores to use for the whole chain of tasks for its own stream. It is coherent to assume that for each stream, there are always enough data to process as we work in Big-Data environment. Third, for the task selection of each stage in exploration mode (or as the authors call it the quality check module), when the observed quality of a stage decreases, a different task is selected for exploration. Once the observed quality of the first stage is optimized, the following stage is then optimized and so on. This last modification is only related to the quality check module as a clear systematic methodology for selecting the tasks was not provided in the literature. Finally, in the reward function, a higher priority is given to the quality.

Since the results obtained for stream 1, 2 and 3 are similar to each other, we only illustrate the results obtained for stream 3. Figure 16 compares the throughput and quality (average value of all the stages accumulated) between our S-MAB and SSP [19]. The figure shows that our solution outperforms the SSP solution in terms of obtained throughput and observed quality. To understand why [19] fails to provide the same level of performance as our proposed solution, in figure 16, we illustrate in details the results obtained for each stage and each data variation phase for stream 3 scheduled with [19]. The SSP algorithm fails for the following reasons:

The algorithm may fall in a local maxima. In fact, the

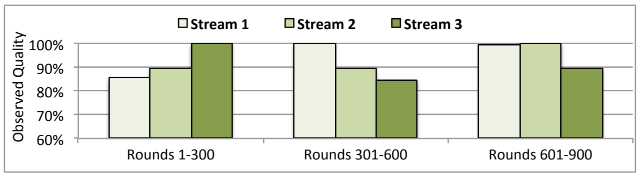


Fig. 13. Evolution of the observed quality for each stream in stage 4 when the minimum required quality set by the user is 80%.

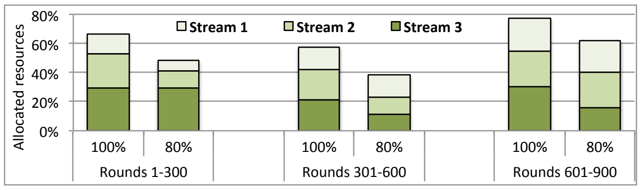


Fig. 14. Comparison of the allocated resources per stream (stacked) in stage 4 between an execution with a minimum quality = 100% and an execution with a minimum quality = 80%).

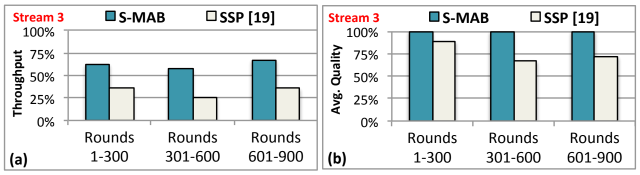


Fig. 15. Performance comparison between S-MAB and SSP [19] for stream 3. (a) The obtained global throughput. (b) The observed average quality (all stages).

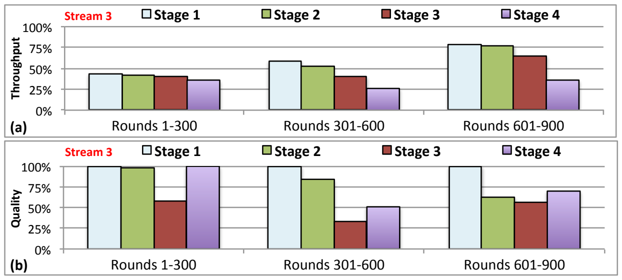


Fig. 16. Evolution of the obtained (a) throughput and (b) quality for stream 3 scheduled with SSP [19].

exploration may find the task that provides the maximum quality for stage 1 but the output of that task is not optimized for the remaining stages. Thus, the algorithm keeps continuously tuning the remaining stages to optimize the quality without reaching the target quality as the first stage is stuck at a local maxima. This is illustrated in Figure 16 (b) in all data variation phases. However, when the task selection is tuned, it may have high impact on the workload resulting in a significant throughput drop. In fact, if selected task in stage 2 is adjusted and requires a higher number of cores than the initial action, then remaining stages may end up with zero core, and data in stage 3 and 4 are discarded. This is illustrated in the results of Figure 16 (b) especially for rounds 300-900. This problem is due to the fact that the scheduler proposed in [19] only controls the throughput but not the cores allocation unlike our new proposed solution which instead learns the cores allocation based on the observed throughput. The throughput has to be a metric that is observed but not controlled.

In terms of the resources required for the execution of [19], the algorithm uses a significant amount of memory compared to our proposed FAL algorithm. In fact, our solution requires only few megabytes to store the observed feedback database, generated action database and the structure holding the different counters. However, in [19], for each action in the action set, a reward matrix and a transition probability matrix of the size of the number of states are allocated. For instance for the setting used in this experiment, there are 75 actions (i.e., 25 throughput values x 3 tasks options) and around 1000 states observed which result in 150 matrices each with a size of 1000x1000 for each stream. Each value in each matrix is stored on 8 bytes (Double precision). Therefore, the total space required to store at least these matrices is over 1GB for each stream. This space can significantly increase when more states are observed or more actions are added. Finally, the algorithm responsible for computing the scheduling policy suffers from a high complexity as the value iteration algorithm goes through all these stored data in order to compute the scheduling policy.

6.2.5 Energy consumption: comparison with an optimal scheduling solution

Since the solution proposed in [19] failed to provide the same level of performance as our proposed solution, we compare our S-MAB algorithm to a second scheduling solution that we have developed and has full knowledge of all the tasks qualities and the workload of each input data

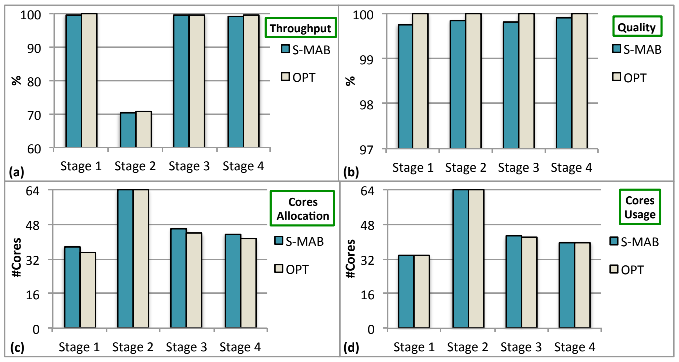


Fig. 17. Performance comparison between S-MAB and a scheduler with full knowledge of the input steams. (a) Obtained throughput. (b) Observed quality. (c) Number of allocated cores. (d) Resource usage.

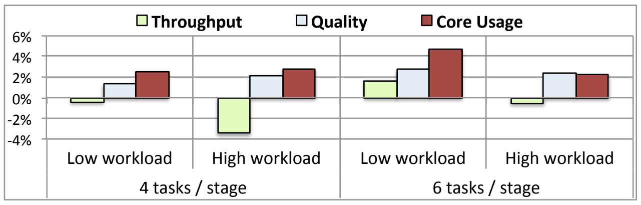


Fig. 18. Synthetic application with different workload and stage configurations: performance comparison of the S-MAB versus a scheduler having full knowledge of the input steams.

for each task in each stage before any scheduling decision is taken. This scheduler provides an optimal solution that guarantees highest quality with the minimum workload and the minimum cores allocations. We call it the optimal scheduler. Figure 17 compares the performance of our S-MAB scheduler to the optimal scheduler namely, (a) the obtained throughput, (b) the observed quality, (c) number of allocated cores and (c) the amount of resource usage.

Figure (a) shows that the throughput value obtained with S-MAB is at most 0.6% less than the obtained optimal value while Figure (b) shows that the observed quality is at most 0.3% less than the obtained optimal value. Moreover, Figure (c) and (d) shows that our algorithm is also energy efficient as it allocates the minimum number of cores required for optimal throughput and optimal quality. In fact Figure (c) shows that the S-MAB algorithm allocates around 4% more resources (i.e, 1 to 3 cores in total) compared to the optimal solution, which is mainly due to the variation of the workload of the photos in each of the 3 input streams. Finally the resource usage is kept at a similar level to the optimal scheduler which confirm the optimized selection of the processing tasks at each stage.

Finally, in order to generalize our obtained results, we have added new synthetic applications with different configurations of processing paths to model multiple types of workloads and different quality requirements. We have generated them following a set of predefined rules to cover exhaustively the possible options of the design space of Big Data applications. Figure 18 shows how far is S-MAB from the optimal solution when varying the number of tasks per stage and the workload intensity. The figure shows a difference in quality around 2%. Moreover, the average throughput obtained with S-MAB can be higher (i.e., negative values in the figure) because of the exploration part (also less quality may generate higher throughput). Finally, the difference in core usage (percentage of used allocated cores) is around 4%.

7 Conclusion

In this paper, we have proposed a new systematic and efficient methodology and associated algorithms for online learning and energy-efficient scheduling of Big-Data streaming applications with multiple streams on many core systems with resources constraints. The key contributions of this work are as follows: (1) We formalized the problem of multi-streams scheduling as a staged decision problem in which the performance obtained for various resource allocations is unknown a priori but learned over time. (2) Our scheduler is able to determine which processing method to assign to each stream and how to allocate resources over time in order to maximize the performance on the fly, at run-time, without having access to any offline information.

(3) Unlike other online learning methods such as standard multi-armed bandits and reinforcement learning, in our formulation the outcome of each scheduling action depends on a sequence of previous scheduling decisions and feedbacks that are taken at a certain stage (window) of time.

References

[1] R. Ducasse, *et al.*, "Adaptive topologic optimization for large-scale stream mining," in *IEEE JSTSP*, vol. 4, no. 3, pp. 620–636, June 2010.
[2] F. Fu, *et al.*, "Configuring competing classifier chains in distributed stream mining systems," in *IEEE JSTSP*, vol. 1, no. 4, pp. 548–563, December 2007.
[3] B. Foo, *et al.*, "A distributed approach for optimizing cascaded classifier topologies in teal-time stream mining systems," in *IEEE TIP*, vol. 19, no. 11, pp. 3035–3048, November 2010.
[4] B. Foo, *et al.*, "Configuring trees of classifiers in distributed multimedia stream mining systems," in *IEEE TCSVT*, vol. 21, no. 3, pp. 245–258, March 2011.
[5] D.S. Turaga, *et al.*, "Resource management for networked classifiers in distributed stream mining systems," in Proc. *ICDM*, 2006.
[6] N. Tatbul, *et al.*, "Load shedding in a data stream manager," in Proc. *VLDB*, 2003.
[7] Y. Chi, *et al.*, "Loadstar: Load shedding in data stream mining," in Proc. *VLDB*, 2005.
[8] J. Xu, *et al.*, "Learning optimal classifier chains for real-time Big-Data mining," in Proc. *Annual Allerton Conference*, 2013.
[9] K. Kanoun, *et al.*, "Low power and scalable many-core architecture for Big-Data stream computing," in Proc. *ISVLSI*, 2014.
[10] M. L. Puterman, "Markov decision processes: Discrete stochastic dynamic programming," John Wiley and Sons, New York, NY, 1994.
[11] J. Gama, *et al.*, "A survey on concept drift adaptation," *ACM Computing Surveys*, 2014.
[12] C. Ballard, *et al.*, "IBM InfoSphere streams. Harnessing data in motion," IBM Redbooks 2010.
[13] J. G. Koomey, *et al.*, "Estimating total power consumption by servers in the U.S. and the world," Stanford Univ. Press, 2007.
[14] P. Auer, *et al.*, "Finite-time analysis of the multiarmed bandit problem," in *Machine Learning*, vol. 47, pp. 235–256, May-June 2002.
[15] T. Lai, *et al.*, "Asymptotically efficient adaptive allocation rules," in *Advances in Applied Mathematics*, vol. 6, pp. 4–22, 1985.
[16] Y. Gai, *et al.*, "Combinatorial network optimization with unknown variables: Multi-armed bandits with linear rewards and individual observations," in *IEEE TON*, vol. 20, no. 5, pp. 1466–1478, 2012.
[17] A. Slivkins, "Contextual bandits with similarity information," in Proc. *COLT*, 2011.
[18] N. Cesa-Bianchi, *et al.*, "Combinatorial bandits," in *JCSS*, vol. 78, no. 5, pp. 1404–1422, 2012.
[19] K. Kanoun, *et al.*, "Big-Data streaming applications scheduling with online learning and concept drift detection" in Proc. *DATE*, 2015.
[20] Y. Matsumoto, *et al.*, "Manycore processor for video mining applications," in Proc. *ASP-DAC*, 2013.
[21] L. Schor, *et al.*, "Reliable and efficient execution of multiple streaming applications on Intels SCC processor," in Proc. *ROME*, 2013.
[22] Tilera TILE-Gx72. http://www.tilera.com
[23] R. Schöne, *et al.*, "Wake-up latencies for processor idle states on current x86 processors," in *CSRD*, 2014.
[24] Intel Xeon Processor 5500 series datasheet, http://www.intel.com
[25] "OpenCV," http://opencv.org
[26] P. Viola, *et al.*, "Rapid object detection using a boosted cascade of simple features," in Proc. *IEEE CVPR*, 2001.
[27] A.S. Georghiades, *et al.*,"From few to many: Illumination Cone models for face recognition under variable lighting and pose," in *IEEE TPAMI* vol. 23, no. 6, pp. 643–660, 2001.
[28] S. Milborrow, *et al.*,"The MUCT landmarked face database," in Proc. *PRASA*, 2010.
[29] "Faces 1999 (Front)," http://www.vision.caltech.edu/archive.html
[30] "The BioID face database," https://www.bioid.com/About/BioID-Face-Database
[31] C. Tekin, *et al.*, "Online learning of rested and restless bandits," in *IEEE Trans. Inf. Theory* vol. 58, no. 8, pp. 5588–5611, 2012.
[32] K. Liu, *et al.*, "Distributed learning in multi-armed bandit with multiple players," in *IEEE Trans. Signal Process.* vol. 58, no. 11, pp. 5667–5681, 2010.
[33] A. Anandkumar, *et al.*, "Distributed algorithms for learning and cognitive medium access with logarithmic regret," in *IEEE JSAC* vol. 29, no. 4, pp. 731–745, 2011.
[34] A. Tewari and P. Bartlett, "Optimistic linear programming gives logarithmic regret for irreducible MDPs," in *Advances in Neural Information Processing Systems* vol. 20, pp. 1505–1512, 2008.
[35] P. Auer, *et al.*, "Near-optimal regret bounds for reinforcement learning," in *Advances in Neural Information Processing Systems* pp. 89–96, 2009.
[36] R. Ortner, *et al.*, "Regret bounds for restless Markov bandits," in *Algorithmic Learning Theory* pp. 214–228, 2012.
[37] E. Even-Dar, Y. Mansour, "Learning rates for Q-learning," in *The Journal of Machine Learning Research* vol. 5, pp. 1–25, 2004.
[38] M. Brezzi, T.Z. Lai, "Optimal learning and experimentation in bandit problems," in *Journal of Economic Dynamics and Control* vol. 27, no. 1, pp. 87–108, 2002.
[39] V. Gabillon, *et al.*, "Adaptive submodular maximization in bandit setting," in *Advances in Neural Information Processing Systems* pp. 2697–2705, 2013.
[40] D. Golovin, A. Krause, "Adaptive submodularity: A new approach to active learning and stochastic optimization," in *COLT*, pp. 333–345, 2010.
[41] C.H. Papadimitriou, J.N. Tsitsiklis, "The complexity of optimal queuing network control," in *Math. Oper. Res.*, vol. 24, no. 2, pp. 293-305, 1999.

Karim Kanoun received his Ph.D. degree in Electrical Engineering at École Polytechnique Fédérale de Lausanne (EPFL), Switzerland in 2015 and his MSc degree in Computer Science from the ENSIMAG school of engineering in informatics and applied mathematics in Grenoble, France in 2009. He is currently a Post-Doctoral Researcher in the Embedded Systems Laboratory (ESL) at EPFL. His research interests include machine learning and energy efficient schedulers for Big-Data stream mining applications on mobile many-core platforms and large scale systems.

Cem Tekin (S'09-M'13) is an Assistant Professor in Electrical and Electronics Engineering Department at Bilkent University, Turkey. From 2013 to 2015, he was a Postdoctoral Scholar at UCLA. He received Ph.D. degree in electrical engineering: systems from the University of Michigan, Ann Arbor, in 2013. His research interests include machine learning and data mining.

David Atienza (M'05-SM'13-F'16) is an associate professor of EE, and director of the ESL at EPFL. His research interests include system-level design methodologies for both high- and low-end multi-processor system-on-chip (MPSoC) and embedded systems, including new 2-D/3-D thermal-aware design for MPSoCs, ultra-low power system architectures for wireless body sensor nodes, HW/SW reconfigurable systems, dynamic memory optimizations, and network-on-chip design. He is a co-author of more than 200 publications, and five U.S. patents. He was Technical Programme Chair of IEEE/ACM DATE 2015, and received the IEEE CEDA Early Career Award in 2013, the ACM SIGDA Outstanding New Faculty Award in 2012 and has been Distinguished Lecturer (period 2014-2015) of IEEE CASS. He is an IEEE Fellow and Senior Member of ACM.

Mihaela van der Schaar is Chancellor's Professor of Electrical Engineering at University of California, Los Angeles. She is an IEEE Fellow, was a Distinguished Lecturer of the Communications Society (2011-2012), the Editor in Chief of IEEE Transactions on Multimedia (2011-2013) and a member of the Editorial Board of the IEEE Journal on Selected Topics in Signal Processing (2011). Her research interests include engineering economics and game theory, multi-agent learning, online learning, decision theory, network science, multi-user networking, Big data and real-time stream mining, and multimedia.