EXTRA: An Experience-driven Control Framework for Distributed Stream Data Processing with a Variable Number of Threads

Teng Li*†, Zhiyuan Xu*†, Jian Tang†, Kun Wu†, and Yanzhi Wang◇
†Department of Electrical Engineering and Computer Science, Syracuse University
◇Department of Electrical and Computer Engineering, Northeastern University
†{tli01, zxu105, jtang02, kwu102}@syr.edu ◇yanz.wang@northeastern.edu

***Abstract*—In this paper, we present design, implementation and evaluation of a control framework, EXTRA (EXperience-driven conTRol frAmework), for scheduling in general-purpose *Distributed Stream Data Processing Systems (DSDPSs)*. Our design is novel due to the following reasons. First, EXTRA enables a DSDPS to dynamically change the number of threads on the fly according to system states and demands. Most existing methods, however, use a fixed number of threads to carry workload (for each processing unit of an application), which is specified by a user in advance and does not change during runtime. So our design introduces a whole new dimension for control in DSDPSs, which has a great potential to significantly improve system flexibility and efficiency, but makes the scheduling problem much harder. Second, EXTRA leverages an experience/data driven model-free approach for dynamic control using the emerging Deep Reinforcement Learning (DRL), which enables a DSDPS to learn the best way to control itself from its own experience just as a human learns a skill (such as driving and swimming) without any accurate and mathematically solvable model. We implemented it based on a widely-used DSDPS, Apache Storm, and evaluated its performance with three representative Stream Data Processing (SDP) applications: continuous queries, word count (stream version) and log stream processing. Particularly, we performed experiments under realistic settings (where multiple application instances are mixed up together), rather than a simplified setting (where experiments are conducted only on a single application instance) used in most related works. Extensive experimental results show: 1) Compared to Storm's default scheduler and the state-of-the-art model-based method, EXTRA substantially reduces average end-to-end tuple processing time by 39.6% and 21.6% respectively on average. 2) EXTRA does lead to more flexible and efficient stream data processing by enabling the use of a variable number of threads. 3) EXTRA is robust in a highly dynamic environment with significant workload change.**

I. Introduction

General-purpose *Distributed Stream Data Processing Systems (DSDPSs)* (such as Apache Storm [33] and Spark Streaming [31]) have attracted extensive attention from industry and academia in recent years. They are capable of processing unbounded big streams of continuous data in an distributed and real (or near-real) time manner, which, however, have quite different programming models or runtime systems, compared to MapReduce-based batch processing systems (such as Hadoop [3] and Spark [30]) that deal with static big data in an offline manner. The scheduling problem is a fundamental problem in a DSDPS, i.e., the problem of assigning threads (carrying workload) to workers/machines with the objective of minimizing average end-to-end tuple processing time (which we may simply call tuple processing time). Currently, with most existing methods, a user specifies the number of threads to carry workload for a (logic) Processing Unit (PU) in an application in advance without knowing much about runtime needs, which, however, does not change during runtime. We aim to develop a control framework for scheduling, which is capable of using a variable number of threads on the fly. This design introduces a whole new dimension for control in DSDPSs, which has a great potential to significantly improve system flexibility and efficiency, but makes the scheduling problem much harder.

The current practice of scheduling includes two kinds of approaches : 1) The simple load-balancing approach (such as the default scheduler of Storm [33]): It evenly distributes threads over machines in the cluster with a round-robin manner. 2) The model-based approach (such as [15]): It builds a mathematical model to estimate end-to-end tuple processing time or some other important metrics for a given scheduling solution and then uses the model to guide scheduling. The first approach is obviously not effective due to lack of consideration for runtime states such as communication delays between machines/processes/threads, which have been shown to have a significant impact on end-to-end average tuple processing time [38]. The second approach may not work well too since it is very hard and challenging to mathematically model a DSDPS, which usually has a very complicated architecture, and highly time-variant runtime states. Specifically, queueing theory has been employed to model distributed stream database systems [24]. However, we doubt it can work well for a DSDPS because: 1) The queueing theory still cannot well handle a queueing network (rather than a single queue), while a DSDPS represents a very complicated queueing network with a multi-point to multi-point structure where tuples from a queue is distributed to multiple different downstream queues, and a queue receives tuples from multiple different upstream queues. 2) The whole theory was built based on a few strong

*Both authors contributed equally to this work.

assumptions (e.g, Poisson distribution for the arrival process, etc), which, however, may not hold in a complicated DSDPS (e.g. the tuple arrival process between threads). An interesting model was presented in a recent work [15] to estimate end-to-end tuple processing time in a DSDPS by summing up delay at each component (including processing time at each PU and communication delay between two PUs and predicting such delays using a supervised learning algorithm. A scheduling algorithm was also developed based on estimates provided by this model. Even though it has been shown to outperform the simple load-balancing approach, we suspect that it is still far away from an ideal solution because the prediction for each individual component may not be accurate; and more importantly, end-to-end tuple processing time involves many factors in a complex DSDPS, which are not fully captured by the model. In addition, we are facing a much more challenging scheduling problem of jointly determining the number of threads and their assignment here. None of methods in related works [15], [17] can be directly applied to solve this problem since they only work for the case with a fixed number of threads.

To overcome these issues, we aim to develop a data/ experience model-free approach that can fully embrace the *extra* power and flexibility brought by a variable number of threads, and learn the best way to control a DSDPS from its own experience rather than any accurate and mathematically solvable system model, just as a human learns a skill (such as driving and swimming). We believe that the emerging *Deep Reinforcement Learning (DRL)* [22] is a promising technique for archiving the above goal because 1) it enables model-free control that does not rely on any accurate mathematical model; 2) it can better handle a sophisticated state space (e.g., AlphaGo [29]), compared to traditional Reinforcement Learning (RL) [34]; and 3) it is able to deal with highly time-variant environments. Even though DRL has been shown to deliver superior performance on a series of game-playing applications [22], its applicability on controlling and managing complex distributed computing systems such as DSDPSs remains unknown. The basic DRL technique, such as Deep Q Network (DQN) [22], does not work here since it can only deal with control problems that have a limited action space but the scheduling problem in a DSDPS usually has an enormous action space (worker threads, worker processes, virtual/physical machines, and their combinations, see Section II). When considering the scenario with a variable number of threads, the scheduling problem becomes much harder (See Section II-B) and the correlations between a scheduling solution and the average tuple processing time become more sophisticated, which further justify the necessity of using a learning (especially deep learning) approach. However, it is more challenging because the action space becomes much larger.

To the best of our knowledge, we are among the first to develop a highly-effective experience-driven control framework for scheduling in DSDPSs based on DRL, which minimizes average end-to-end tuple processing time with a variable number of threads. In summary, we made the following contributions:

- We develop a control framework, EXTRA, for a DSDPS, which enables dynamic use of a variable number of threads at runtime; while the methods proposed in most related works (such as [15], [17], [38]) can only work with the case with a fixed number of threads, which therefore cannot directly applied here.
- We conducted a comprehensive performance evaluation and showed via extensive experiments with representative Stream Data Processing (SDP) applications that EXTRA significantly outperforms the commonly-used baseline as well as the state-of-the-art. It is worth mentioning that we performed experiments under realistic settings (where multiple application instances are mixed up together), instead of a simplified setting (where experiments are conducted only on a single application instance) used in most related works (such as [15], [17], [38]).

II. Design and Implementation of the Proposed Framework

A. Overview

In a DSDPS, logically, a Processing Unit (PU) is used to consume and process tuples from external data sources (or other PUs). A task is defined as an instance of a data source or a PU. For each data source or PU, multiple tasks can be created and executed in parallel on a cluster. Physically, a task is normally mapped to a worker thread at runtime, which processes incoming data tuples using user code. A DSDPS usually runs on a computer cluster with multiple physical or virtual machines, and a master node serving as the central control unit, which handles the distribution of user code over the cluster, scheduling and fault tolerance. Each machine can be used to host multiple worker processes, and each worker process can host multiple worker threads.

In a DSDPS, a scheduling solution specifies the assignment of worker threads to worker processes, and worker processes to machines. Current DSDPSs typically include a default scheduler, which can be replaced by a custom scheduler. The default scheduler usually applies a simple scheduling algorithm, which evenly distributes workload by assigning a pre-configured number of worker threads to worker processes and further to machines in a round-robin manner.

To enable dynamic use of a variable number of threads at runtime and realize online scheduling in a DSDPS, we present a DRL-based experience-driven control framework, EXTRA, which mainly consists of three components:

1) *DRL Agent:* As the central part of EXTRA, this component applies a DRL-based method with the input of a state, produces an action (Section II-B) and translates it into corresponding scheduling solution, which is further pushed to the custom scheduler.
2) *Data Store:* This component leverages a database for storing transition samples (for the training purpose), including information about the state, action and reward (Section II-B).

3) *Custom Scheduler:* Upon receiving a scheduling solution from DRL agent, this component deploys it on the DSDPS via its master, which specifies the number of threads of each PU or data source, and their assignment.

The design of our framework leads to several desirable features, which makes EXTRA superior to existing methods. First, EXTRA is capable of varying the number of threads for each PU during runtime, which leads to more flexible and effective control thus offers a DSDPS an extra leverage to better handle a highly dynamic environment. This is a desirable feature that most scheduling methods (such as [15], [17], [38]) do not have. For most of them (including Storm's default scheduler), in an application, the number of threads of each PU is pre-defined by its user with limited or almost no knowledge about runtime states and demands, and remains unchanged throughout the whole data processing procedure. Second, our design features experience-driven model-free control powered by emerging DRL (Section II-B), which gradually discovers the best way to control a DSPDS from its own experience; while the current practice employs a rather trivial method that equally distributes workload over all available machines; and some existing optimization-based methods (e.g. [16]) perform online scheduling based on mathematical models that may not able to accurately characterize a complex DSDPS at runtime. In addition, we strongly emphasize user transparency (Section II-C) in our design so that a user do not need to make any changes to their original code in order to run their applications on the new DSDPS with EXTRA.

B. DRL-based Scheduling

In this section, we present the proposed DRL-based scheduling algorithm in EXTRA, which targets at minimizing the end-to-end average tuple processing time via jointly determining the number of threads and their assignment. First, we summarize the major notations below for quick reference.

TABLE I
MAJOR NOTATIONS

Notation	Description
$\mathcal{M}$ and M	The set of machines and the total number of machines
$\mathcal{P}$	The set of processes
$\mathcal{N}$ and N	The set of threads and the maximum number of threads
$\mathcal{C}$ and C	The set of components and the total number of components
C_i and c_i	The maximum and actual number of threads from component i
H	The number of application instances
$\mathbf{s}$ and $\mathcal{S}$	The state and state space
$\mathbf{a}, \hat{\mathbf{a}}$ and $\mathcal{A}$	The action, proto-action and action space
r	The reward
$\boldsymbol{\theta}$ and $\boldsymbol{\phi}$	Weights of actor and critic networks $p(\cdot)$ and $Q(\cdot)$

Suppose that we are given a set of machines $\mathcal{M}$, processes $\mathcal{P}$ and threads $\mathcal{N}$, a scheduling solution specifies the mappings of $\mathcal{N} \mapsto \mathcal{P}$ and $\mathcal{P} \mapsto \mathcal{M}$, which represents the assignment of each thread to a process of a machine. Similar as [38], [15], our design ensures that threads from the same application instance are assigned to only one process on a machine, which leads to better performance than the solution that may assign them to more than one process due to additional but unnecessary inter-process communications [38]. Based on this design, we only need to specify how to assign each thread to a machine, i.e., mapping of $\mathcal{N} \mapsto \mathcal{M}$. Note that we call it application instance (i.e., topology in Storm) instead of application because there can be more than one concurrent instances of a common application in a DSDPS.

We consider a realistic scenario where there are H application instances running on a DSDPS simultaneously, which consist of a set of components $\mathcal{C}$ (data sources or processing units). C is the total number of components and c_i is the actual number of active threads of component i, which is bounded by a maximum value of C_i. So $c_i \in \{1, \cdots, C_i\}$ since each component needs to have at least one thread.

The problem of scheduling with a variable number of threads seeks a solution that specifies how many threads of each component in $\mathcal{C}$ is assigned to each machine in $\mathcal{M}$. Note that this problem is much harder than that in [17]. According to [15], because of different tuple processing time and transfer delays between threads at runtime, different scheduling solutions may lead to different end-to-end tuple processing times There are several state-of-the-art DRL methods has been proposed before in [22], [18], [5], particularly, here we introduce how to leverage DDPG [18] for solving the above scheduling problem. We first define the state, action and reward as follows.

ACTION: An action is defined as $\mathbf{a} =< a_{ij} >, \forall i \in \{1, \cdots, C\}, \forall j \in \{1, \cdots, M\}$, where $a_{ij} \in \{0, \cdots, C_i\}, \forall i, j$ and $1 \leq \sum_{j=1}^{M} a_{ij} \leq C_i, \forall i$, and $a_{ij} = a$ means assigning a threads of component i to machine j. The constraints ensure that the total number of threads from each component does not exceed the corresponding maximum value, and each component needs to have at least one thread. Note that a_{ij} could be 0, which means no thread of component i is assigned to machine j. The action space $\mathcal{A}$ contains all feasible actions, whose size is exponentially large.

STATE: A state $\mathbf{s} = (\mathbf{a}, \mathbf{w})$ consists of two parts: the current thread assignment $\mathbf{a} =< a_{ij} >$ as described above, and the workload $\mathbf{w}$. Here, $\mathbf{w} = [w_1, \cdots, w_h, \cdots, w_H]$, where w_h is the tuple arrival rate (the number of tuples per second) of the hth application instance.

REWARD: The reward is defined to be the negative value of the sum of average tuple processing time over all running application instances so that maximizing this reward is equivalent to minimizing the sum of average tuple processing times. Note that weights can be assigned to application instances to indicate their priorities, then the reward becomes the weighted sum of the average tuple processing times.

It is worth mentioning that the success of DRL methods highly depends on the definition of state space, action space and reward. In our problem, the definition of action space and reward are relatively straightforward. For the state space, there are many different ways available because a DSDPS includes various runtime information (features) [15], e.g. average tuple processing latency at each thread/PU, average tuple transfer latency between threads/PUs, workload at each thread/process/machine, CPU/memory/network usages of machines, etc. We, however, choose a simple design. Among

various runtime information, we pick the scheduling solution $\mathbf{a}$, which leads to different values for other features (mentioned above) and eventually different average end-to-end tuple processing times, and the incoming tuple rates at the data sources $\mathbf{w}$ reflect the real-time workload of the DSDPS, which is also necessary and meaningful. According to our observation, the proposed DNNs can well model correlations between a state and a reward after training. Our design of the state space can significantly reduce control overhead since only very limited runtime information needs to be collected at each decision epoch, which can be easily done in a DSDPS (e.g., by using the Storm REST API).

The basic DRL technique, the DQN-based method [22], adopts a value iteration approach, taking state $\mathbf{s}$ and action space $\mathcal{A}$ as input and return value $Q = Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\theta})$ for each action $\mathbf{a} \in \mathcal{A}$ (with parameters $\boldsymbol{\theta}$). A greedy method can then be applied to select an action in each epoch. In order to apply this method, the action space $\mathcal{A}$ needs to be restricted to be polynomial-time searchable. However, the action space here is exponentially large. Hence, we don't see a straightforward extension to address our problem.

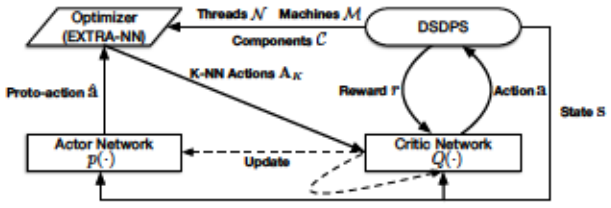


Fig. 1. The actor-critic-based method

In our design, we take some state-of-the-art RL techniques, including actor-critic method [34], [5] and the deterministic policy gradient method [28], to solve the scheduling problem in a DSDPS. Note that since these techniques only provide a general framework, none of them can be directly applied to a DSDPS, we still need to carefully design a specific solution to our problem. The overall architecture of the proposed method is presented in Figure 1, which mainly consists of three components: an actor network, an optimizer and a critic network. In short, the actor network takes the state as input and outputs a continuous *proto-action* $\hat{\mathbf{a}}$ (base solution), the optimizer derives a set of K-Nearest Neighbors (K-NN) $\mathbf{A}_K$ of a given proto-action $\hat{\mathbf{a}}$ in the action space, and the critic network takes the state and K-NN actions $\mathbf{A}_K$ as input and outputs the Q value for each action $\mathbf{a} \in \mathbf{A}_K$.

Specifically, the actor network $p(\mathbf{s}|\boldsymbol{\theta})$ is a function with parameters $\boldsymbol{\theta}$. Since the output of the actor network $\hat{\mathbf{a}}$ (e.g. the proto-action) is continuous, we can know that $\hat{\mathbf{a}} \notin \mathcal{A}$ (i.e., not a feasible action) thus it can not be directly considered as a scheduling solution.

The next and most critical step is to derive the K-NN of a given proto-action, which has not been well discussed in previous works, like [5]. It is known that finding K-NN can easily be done in linear time. However, the input size is the total number of actions here, which is exponentially large. Therefore, it may take an exponentially long time to enumerate all actions and do a linear search for the set of K-NN. In order to limit the searching time, we propose an optimizer based on Mixed-Integer Quadratic Programming (MIQP), which is defined as:
EXTRA-NN:

$$\begin{aligned}
&\min_{\mathbf{a}}: \quad \|\mathbf{a} - \hat{\mathbf{a}}\|_2^2 \\
&\text{s.t.:} \sum_{j=1}^{M} a_{ij} \geq 1, \forall i \in \{1, \cdots, C\}; \\
&\quad \sum_{j=1}^{M} a_{ij} \leq C_i, \forall i \in \{1, \cdots, C\}; \\
&\quad a_{ij} \in \{0, \cdots, C_i\}, \forall i \in \{1, \cdots, C\}, \forall j \in \{1, \cdots, M\}.
\end{aligned} \tag{2.1}$$

Solving EXTRA-NN can find the nearest neighbor $\mathbf{a}$ of the proto-action action $\hat{\mathbf{a}}$. The constraints ensure the feasibility of action $\mathbf{a}$, i.e., $\mathbf{a} \in \mathcal{A}$ (See the definition of an action). K-NN of $\hat{\mathbf{a}}$ can be obtained by iteratively solving EXTRA-NN K times. In each iteration, one neighbor of the proto-action is obtained and the corresponding $< a_{ij} >$ is fixed, then MIQP-NN is updated and solved again to find the next nearest neighbor. In practical applications, since C, $< C_i >$ and M are not too large, the MIQP problem can be efficiently solved. In our experiments, we solved EXTRA-NN using the Gurobi Optimizer [12] and found that solving a problem instance could usually be done in real time (within about 10ms using a regular desktop). For very large input cases, we can improve computational efficiency by relaxing the EXTRA-NN problem to a convex programming problem [9] and using a rounding algorithm to obtain approximate solutions.

The last step is to select an action from $\mathbf{A}_K$ using the critic network. The critic network $Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\phi})$ is a function with parameters $\boldsymbol{\phi}$, which outputs Q value for each action $\mathbf{a} \in \mathbf{A}_K$. The feasible action can be selected by:

$$\pi_Q(\mathbf{s}) = \underset{\mathbf{a} \in \mathbf{A}_K}{\operatorname{argmax}} Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\phi}). \tag{2.2}$$

We formally present the DRL-based algorithm for scheduling as Algorithm 1. Both experience replay and target networks [22], [18] are adopted in our algorithm to improve learning stability and avoid divergence. For experience replay, samples are first stored into a replay buffer $\mathbf{B}$, then a mini-batch of transition samples are randomly selected from $\mathbf{B}$ to train the actor and critic networks, instead of using immediately collected transition sample at each decision epoch t (lines 13–14). Note that the oldest transition sample will be discarded when $\mathbf{B}$ exceeds its maximum capacity. Target networks have the same network structure as the original actor network or critic network, but their weights $\boldsymbol{\theta}'$ and $\boldsymbol{\phi}'$ are slowly updated by a parameter τ (line 19). In our implementation, we set the

size of experience replay buffer $|\mathbf{B}| = 1000$ and mini-batch $L = 32$. τ is set to 0.01.

Algorithm 1 The DRL-based scheduling algorithm

1: Randomly initialize actor network $p(\cdot)$ and critic network $Q(\cdot)$ with weights $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ respectively;
2: Initialize target networks $p'(\cdot)$ and $Q'(\cdot)$ with weights $\boldsymbol{\theta'} \leftarrow \boldsymbol{\theta}$, $\boldsymbol{\phi'} \leftarrow \boldsymbol{\phi}$; /**Offline Training**/
3: Initialize and load the historical transition samples into experience replay buffer $\mathbf{B}$, pre-train $p(\cdot)$ and $Q(\cdot)$ offline; /**Online Learning**/
4: **for** episode = 1 **to** E **do**
5: Initialize a random process $\mathcal{X}$ for exploration;
6: Receive an initial observation state $\mathbf{s}_1$; /**Decision Epoch**/
7: **for** t = 1 **to** T **do**
8: Derive proto-action $\hat{\mathbf{a}}$ from $p(\cdot)$;
9: Apply exploration policy to $\hat{\mathbf{a}}$: $\mathcal{X}(\hat{\mathbf{a}}) = \hat{\mathbf{a}} + \epsilon\mathbf{I}$;
10: Obtain K-NN actions $\mathbf{A}_K$ of $\hat{\mathbf{a}}$ from EXTRA-NN;
11: Select action $\mathbf{a}_t = \mathrm{argmax}_{\mathbf{a}\in\mathbf{A}_K} Q(\mathbf{s}_t, \mathbf{a})$;
12: Deploy the scheduling solution according to $\mathbf{a}_t$, then observe the reward and the new state;
13: Store transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ into $\mathbf{B}$; /**Updating the networks**/
14: Randomly sample a mini-batch of transition samples $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ with size L;
15: Obtain K-NN $A_{i+1,K}$ of $p'(\mathbf{s}_{i+1})$ from EXTRA-NN;
16: Compute target values by Equation 2.3;
17: Update $\boldsymbol{\phi}$ to minimize the loss by Equation 2.4:
18: Update $\boldsymbol{\theta}$ with the sampled gradient by Equation 2.5:
19: Update $\boldsymbol{\theta'}$ and $\boldsymbol{\phi'}$ as follows: $\boldsymbol{\theta'} := \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta'}$, $\boldsymbol{\phi'} := \tau\boldsymbol{\phi} + (1-\tau)\boldsymbol{\phi'}$;
20: **end for**
21: **end for**

The actor and critic networks are first pre-trained in an offline manner using historical transition samples (line 3). By introducing pre-training, more possible states and actions can be explored so that online learning can be accelerated. The pre-training process is almost the same as the online learning procedure (lines 14–19). In our implementation, for each experiment setup, $10,000$ transition samples were first collected with random actions. The actor and critic networks were then pre-trained using those samples. In our implementation, the actor network consists of 2-layer full-connected neural network, which contains 64 and 32 neurons with the hyperbolic tangent activation function respectively. The same neural network structure is applied to the critic network.

An online exploration policy is applied to the proto-action $\mathcal{X}(\hat{\mathbf{a}}) = \hat{\mathbf{a}} + \epsilon\mathbf{I}$ (line 9), where ϵ is a decay parameter and $\mathbf{I}$ is a random noise sampled from $[0, 1]$ uniformly in our implementation. Like the ϵ-greedy method in [27], rather than directly taking the derived action from the actor network, ϵ determines the probability of adding a random noise to the proto-action. Moreover, ϵ decreases with decision epoch t, that is, as the training continues, it is more likely that derived actions will be taken instead of random ones. A trade-off between exploration and exploitation is thus achieved.

To train the actor and critic network, we first sample a mini-batch of transition samples from $\mathbf{B}$ (line 14); for each transition sample $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$, we can find the K-NN actions $\mathbf{A}_{i+1,K}$ of the proto-action $p'(\mathbf{s}_{i+1})$ from the solution of EXTRA-NN (line 15). Note that the proto-action $p'(\mathbf{s}_{i+1})$ for the next state $\mathbf{s}_{i+1}$ is derived from the target actor network $p'(\cdot)$ instead of the original one; then, the target values for the critic network can be calculated by (line 16):

$$y_i := r_i + \gamma \max_{\mathbf{a}\in\mathbf{A}_{i+1,K}} Q'(\mathbf{s}_{i+1}, \mathbf{a}), \tag{2.3}$$

finally, the critic network can be trained by the loss function (line 17), which is defined by a common used mean squared error:

$$\mathcal{L}(\boldsymbol{\phi}) = \frac{1}{L}\sum_i [y_i - Q(\mathbf{s}_i, \mathbf{a}_i)]^2, \tag{2.4}$$

and the actor network is trained by the gradients from the critic network and the chain rule (line 18), which is known as the deterministic policy gradient method [28], [5].

$$\nabla_{\boldsymbol{\theta}} J = \frac{1}{L}\sum_i \nabla_{\mathbf{a}} Q(\mathbf{s}, \hat{\mathbf{a}})|_{\hat{\mathbf{a}}=p(\mathbf{s}_i)} \cdot \nabla_{\boldsymbol{\theta}} p(\mathbf{s})|_{\mathbf{s}=\mathbf{s}_i}. \tag{2.5}$$

C. Implementation Details

The proposed framework, EXTRA, was implemented based on Apache Storm [33]. The DRL agent was implemented as a separate program running independently from Storm. Note that in Storm, data source, PU, application graph, master, worker process and worker thread are called spout, bolt, topology, Nimbus, worker and executor, respectively. We use these terms interchangeably in the following sections.

In Storm, ZooKeeper [40] is used as a coordination service to maintain it's own scheduling solution and distributed synchronization among machines. The master node, Nimbus, provides interfaces to fetch or update Storm's mutable configurations. A Storm topology contains a topology-specific configuration (including the number of threads for each spout/bolt), which will be loaded before the processing starts. When a tuple emitted from a data source successfully traverses all the PUs of an application, a special thread (called acker in Storm) is usually called to acknowledge that it has been fully processed. The end-to-end tuple processing time is the duration between when the data source emits a tuple and when this tuple is acknowledged. Note that we only focus on the processing times of those tuples that are emitted from data sources, which represent the overall end-to-end processing delay on the entire application instance.

At each decision epoch, the DRL agent calls the Storm REST API to remotely obtain the state and reward (Section II-B). Then the proposed DRL-based scheduling algorithm (Algorithm 1) is applied and the action is generated by the DRL agent. The action is further translated into the corresponding Storm-recognizable scheduling solution, which includes the desirable number of executors and their assignment. The scheduling solution is then pushed to the custom scheduler

through a socket between the custom scheduler and the DRL agent, which we implemented for the communication purpose. Running within Nimbus, the custom scheduler has access to various runtime state information of Storm. Upon receiving the scheduling solution from the DRL agent, the custom scheduler changes the number of executors for each component (if needed) using the *rebalance* command from Storm CLI, then updates the number of tasks accordingly to match the number of executors, and re-submits it to the system. After that, the Nimbus starts to deploy the executors to the worker nodes (i.e. machines) according to the scheduling solution. Note that during the deployment, in order to minimize the overhead, we only re-schedule those executors which have different assignments from the previous ones while keeping the rest unchanged (instead of freeing all executors first and then assigning them to the worker nodes one by one from scratch, which is Storm's default way for re-assignment). Moreover, to ensure accurate data collection, the proposed framework waits for several minutes till the captured data stabilizes after a new scheduling solution is applied. The average of 5 consecutive measurements is taken with a 10-second interval in between.

In addition, both offline training and online learning were performed to train the DRL agent in EXTRA. During the offline training, we collected transition samples from our Storm cluster. Note that this only needs to be done once. After offline training, the DRL agent can quickly reach good scheduling solutions and keep improving itself during online learning.

III. Performance Evaluation

A. Experimental Setup

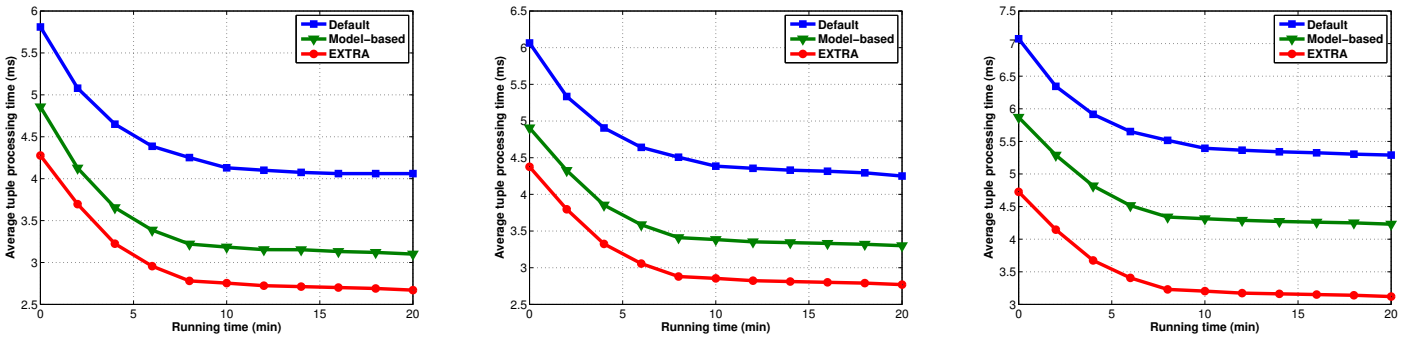
In this section, we present the experimental settings and results of our DRL-based experience-driven control framework (EXTRA). We implemented the whole framework based on Apache Storm [33], and particularly, we implemented and trained the DNNs with TensorFlow [35]. In order to evaluate the performance of EXTRA, we conducted the experiments on a cluster, which includes 1 Nimbus node and 10 worker nodes. Each node has 4GB memory and an Intel Xeon Quad-Core 2.0GHz CPU.

Similar as in [17], we used 3 representative SDP applications to test EXTRA: continuous queries, word count (stream version) and log stream processing. Due to space limitation, we describe the experimental settings here but omit the description for the functions and structures of the three topologies, which can be found in [15], [17]. As aforementioned, EXTRA collects transition samples through interacting with the environment (e.g., DSDPS), and the DRL agent learns the best scheduling policies from these collected transition samples.

Continuous Queries Topology: This topology represents one of the most popular SDP applications. It is a select query that works by initializing access to a database table in memory and looping over each row to check if there is a hit [6]. The topology consists of one type of spout and two types of bolts. The spout randomly generates queries, emitting and sending them to a Query bolt. The database table was created in the memory of each worker node. Upon receiving queries from the spout, the Query bolt starts to scan over the database table, and if the Query bolt find the matching records, it will emits them to the File bolt. The file bolt will writes those records into files. In this topology, a database table which contains vehicles' plates and owner information including owner names and SSNs was randomly generated. Besides, random queries were generated to search the database table for speeding vehicles (with vehicle speeds randomly generated and attached to every entry) and their owners' information. To perform a comprehensive evaluation, we proposed 3 different scenarios: small-scale, medium-scale and large-scale. In every setup, 3 continuous queries topologies (i.e., application instances) were configured to run simultaneously. In the small-scale experiment, for each topology, the maximum number of executors were set to $N = 20$, including a maximum number of 2 spout executors (i.e., $C_1 = 2$), 9 Query bolt executors and 9 File bolt executors respectively. In the medium-scale and large-scale experiments, for each topology, we had a maximum number of 50 and 100 executors in total, including a maximum number of 5 and 10 spout executors, 25 and 45 Query bolt executors and 20 and 45 File bolt executors respectively.

Word Count Topology (stream version): Widely known as a classical MapReduce application, the original version of the topology counts every word's frequency of occurrence in one or multiple files. We modified it into an SDP application running a similar processing routine but with a stream version data source. One type of spout and three types of bolts form a chain-like topology. We used LogStash [21] to read data from input source files. LogStash submits lines of the input file as separate JSON values to a Redis queue. Then they are further consumed and emitted into this topology. We used the text file of Alice's Adventures in Wonderland [1] as the input file. The spout produces a data stream when the input file is pushed into the Redis queue, which is first directed to the SplitSentence bolt. This bolt splits each input line into individual words and further sends them to the WordCount bolt which counts the frequency of occurrence. Finally, the Database bolt stores the results into a Mongo database. In the experiment, 3 word count topologies were configured to run simultaneously. Each topology was configured to have a maximum number of 100 executors in total, including a maximum number of 10 spout executors, 30 SplitSentence bolt executors, 30 WordCount executors and 30 Database bolt executors respectively.

Log Stream Processing Topology: In this topology, log lines are sent to a Redis queue emitting output to the spout. Microsoft IIS log files collected from computers at our university were used as the input data. Based on the rule, the LogRules bolt performs analysis on the log stream, and then delivered values that contains a pre-defined type of log entry instance. The analysis results are delivered to 2 different bolts simultaneously: Counter bolt is used to perform counting actions on log entries; Indexer bolt is used to perform indexing actions. To include two separate database bolts after the



(a) Small-scale (b) Medium-scale (c) Large-scale

Fig. 2. Average tuple processing time over 3 continuous queries topologies

Counter and Indexer bolt respectively, we modified the original topology [2]. Results are stored into separate collections in a Mongo database for the purpose of verification. In the experiment, 3 log stream processing topologies were configured to run simultaneously. Each topology was configured to have a maximum number of 100 executors in total, including a maximum number of 10 spout executors, 20 LogRules bolt executors, 20 Indexer bolt executors, 20 Counter bolt executors and 15 executors for each Database bolt.

Hybrid Scenario: We also came up with a hybrid scenario with 3 different topologies running simultaneously in the large-scale setting. Specifically, there were a continuous queries topology with a maximum number of 100 executors (including a maximum number of 10 spout executors, 45 Query bolt executors and 45 File bolt executors); a word count topology (stream version) with a maximum number of 100 executors (including a maximum number of 10 spout executors, 30 SplitSentence bolt executors, 30 WordCount executors and 30 Database bolt executors); and a log stream processing topology with a maximum number of 100 executors (including a maximum number of 10 spout executors, 20 LogRules bolt executors, 20 Indexer bolt executors, 20 Counter bolt executors and 15 executors for each Database bolt) in our experiment.

Significant Workload Change Scenario: To test how EXTRA performs in a highly dynamic environment, we also performed experiments with significant workload change: the incoming data rate was increased significantly by 50% at 20 minute. The experiments were conducted with 3 continuous queries topologies (the settings were the same as those in the large-scale continuous queries scenario), as well as with 3 different topologies (the settings were the same as those in the hybrid scenario).

B. Experimental Results and Analysis

In this section, we present the experimental results and analysis. To well justify the effectiveness of our framework, we compared EXTRA with a state-of-the-art model-based method proposed in a recent paper [15] (labeled as "Model-based") and the default scheduler of Storm (labeled as "Default") in terms of average (end-to-end) tuple processing time. Both the model-based method and the default scheduler used the maximum number of threads in all the experiments. Note that we could not compare EXTRA with the method presented in [17] in the experiments because it does not work for the case with a variable number of threads. We present the corresponding results in Figures 2–7.

Continuous Queries Topology: As mentioned above, we performed experiments with 3 concurrent continuous queries topologies under 3 settings: small-scale, medium-scale and large-scale, as described above.

As we can see from Figure 2, after deploying a scheduling solution, the average tuple processing time of all 3 methods gradually decreases and eventually gets stable at a relative low value after $8-10$ minutes. For examples, in Figure 2(a), the default scheduler starts at 5.81ms and stabilizes at 4.06ms; the model-based model starts at 4.86ms and stabilizes at 3.10ms; and our EXTRA starts at 4.28ms and stabilizes at 2.67ms. In this case, compared with the default scheduler and the model-based method, EXTRA reduces the average tuple processing time by 34.2% and 14.0% respectively.

From Figure 2(b) (medium-scale), we can see that the average tuple processing times given by all 3 methods slightly go up. Specifically, the default scheduler stabilizes at 4.25ms; while the model-based method stabilizes at 3.31ms; and our EXTRA stabilizes at 2.77ms. Hence, in this case, EXTRA achieves a performance improvement of 34.8% over the default scheduler and 16.2% over the model-based method.

We can observe from Figure 2(c) (large-scale) that the average tuple processing times of all three methods increase further compared to small and medium scale but still can stabilize at acceptable values. This reflects the fact that the workload for the Storm cluster is much heavier but is not overloaded in this large-scale case. Specifically, the default scheduler stabilizes at 5.29ms; while the model-based method stabilizes at 4.23ms; and our EXTRA stabilizes at 3.12ms. In this case, EXTRA achieves a more significant performance improvement of 41.0% over the default scheduler and 26.2% over the model-based method.

Word Count Topology (stream version): We performed

a large-scale experiment over 3 word count (stream version) topologies and show the corresponding results in Figure 3. Since these topologies have the similar complexity to continuous queries topologies, we got a similar average tuple processing time for all 3 methods (compared to that of continuous queries topologies).

In Figure 3, when the default scheduler is used, it stabilizes at 6.51ms; when the model-based method is employed, it stabilizes at 4.57ms; and when EXTRA is used, it stabilizes at 3.37ms. EXTRA results in a significant performance improvement of 48.3% over the default scheduler and 26.3% over the model-based method.

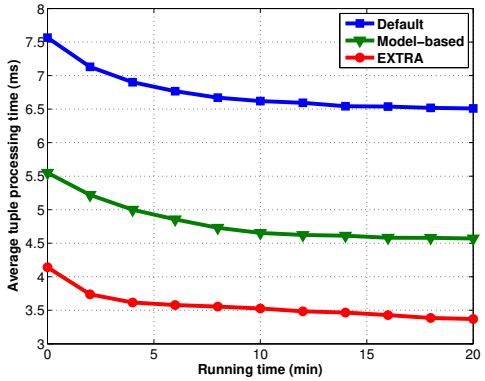


Fig. 3. Average tuple processing time over 3 large-scale word count topologies (stream version)

Log Stream Processing Topology: We performed a large-scale experiment over 3 log stream processing topologies as described above. We show the corresponding results in Figure 4. Since this topology is more complicated compared to the continuous queries topology, it leads to significant longer tuple processing times no matter which scheduler is applied.

In Figure 4, when the default scheduler is used, it stabilizes at 19.93ms; when the model-based method is employed, it stabilizes at 16.15ms; and when EXTRA is used, it stabilizes at 12.33ms. As expected, EXTRA consistently outperforms the other two methods. Specifically, compared to the default scheduler and the model-based method, it reduces the average tuple processing time by 38.1% and 23.6% respectively.

Hybrid Scenario: For the hybrid scenario with 3 different topologies, the corresponding results are shown in Figure 5. When the default scheduler is applied, the average tuple processing time stabilizes at 10.95ms; when the model-based method is applied, it stabilizes at 8.42ms; and when EXTRA is used, it stabilizes at 6.45ms. Hence, EXTRA reduces the average tuple processing time substantially by 41.1% compared to the default scheduler, and 23.4% over the model-based method.

In summary, we can make the following observations from the above results: 1) EXTRA consistently outperforms the other two methods, which well justifies effectiveness of the proposed experience-driven approach as well as the use of a variable number of threads for scheduling in a DSDPS. 2) The

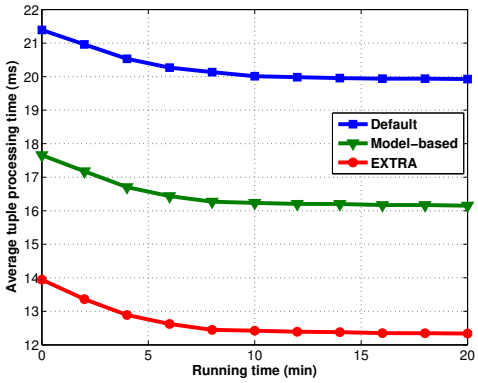


Fig. 4. Average tuple processing time over 3 large-scale log processing topologies

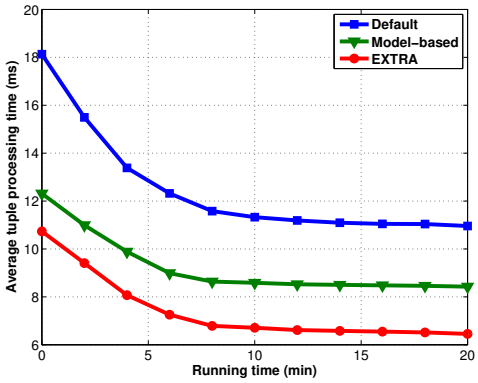


Fig. 5. Average tuple processing time over 3 different large-scale topologies

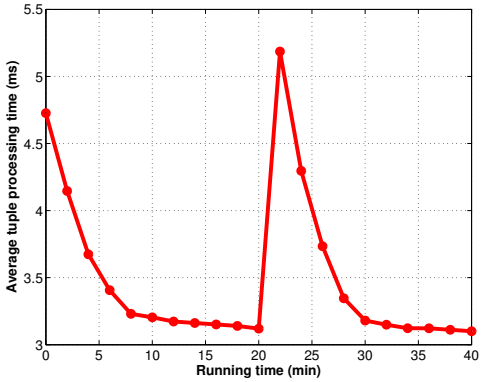


Fig. 6. Average tuple processing time given by EXTRA over 3 large-scale continuous queries topologies with significant workload change

performance improvement offered by EXTRA becomes more and more significant with the increase of the input size, which shows that EXTRA scales very well. 3) Solving EXTRA-NN (Section II-B) can lead to an efficient and effective discovery of the huge action space and thus result in a wise action selection.

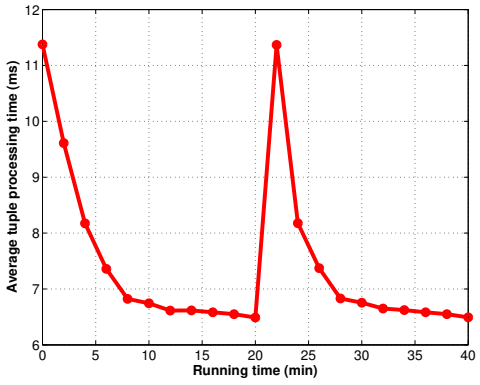


Fig. 7. Average tuple processing time given by EXTRA over 3 different large-scale topologies with significant workload change

Significant Workload Change Scenario: The corresponding results are shown in Figures 6 and 7.

In the first experiment with 3 large-scale continuous queries topologies (Fig. 6), during the first 20-minute period, the average tuple processing time of EXTRA stabilizes at 3.13ms. When the workload increases by 50% at the 20th minute, the result rises sharply to a very high value then gradually stabilizes at 3.15ms respectively. There is a spike because of the sudden adjustment to the scheduling solution. However, once the system gets stable, there is only a very minor increase in the average tuple processing time. Therefore, when there is a significant workload change, EXTRA can quickly adjust the scheduling solution to avoid performance degradation. In the second experiment with 3 different large-scale topologies (Fig 7), similar observations can be made. The initial average tuple processing time is 6.49ms. After the increase of workload, it stabilizes at 6.50ms, which is almost the same as before. These results well justify robustness of EXTRA in a highly dynamic environment with significant workload change.

IV. Related Work

As popular topics in the context of distributed stream database system and DSDPSs, modeling and scheduling have received substantive research attention. In an early work [24], Nicola *et al.* provided a comprehensive survey of performance models (especially queueing models) for distributed and replicated database systems. Active Streams [32] was proposed to enable application-level distributed adaptation, through which, users can dynamically build/extend applications and services by attaching location-independent functional units to the data streams flowing between their applications and network services. Dynamic adaptation techniques were also proposed for distributed continuous query systems [19]. Moreover, Tusch *et al.* [36] presented an adaptive distributed multimedia streaming server architecture (ADMS), which explicitly controls the server-layout. It consists of four types of streaming server components, which all provide dedicated services in an arbitrary number of instances on an arbitrary number of server hosts. Aniello *et al.* [4] proposed two schedulers to improve the performance of Storm: an offline scheduler makes scheduling decisions based on the topology structure; an online scheduler reschedules executors at runtime based on the monitored system states. The authors of [38] presented a traffic-aware scheduling framework, T-Storm, which enables a fine-grained control on worker node consolidation. It aims at minimizing inter-node and inter-process traffic in Storm, and in the meanwhile, it ensures that there is no worker node is overloaded in the system. A similar system called R-Storm (Resource-aware Storm) [25] can satisfy both soft and hard resource constraints as well as minimizing network distance between components that communicate with each other when scheduling tasks. Moreover, Heron *et al.* [13], also developed at Twitter, was created to overcome many of the shortcomings (such as resource isolation, resource efficiency and throughput) that Storm exhibited when running in production at a large scale. The authors of [8] designed and implemented a priority-based resource scheduling method in flow-graph-based DSDPSs. It allows application developers to enhance flow graphs with priority metadata. Jiang *et al.* [14] presented several scheduling strategies for a data stream management system, which aim at minimizing the tuple latency as well as the total memory requirement. They investigated these proposed strategies both theoretically and experimentally. The authors of [37] proposed a prediction-based Quality-of-Service (QoS) management scheme for periodic queries over dynamic data streams. Based on the prediction of the query workload using execution time profiling and input data samplings, the scheme adjusts the query QoS levels. Bedini *et al.* [7] presented a set of models that formalize the performance characteristics of a practical distributed, parallel and fault-tolerant stream processing system that follows the Actor Model theory. They also conducted the experimental validation of the described performance models based on the Storm system. Li *et al.* [15] proposed a predictive scheduling framework to enable fast and distributed stream data processing, which features topology-aware modeling for performance prediction and predictive scheduling. Basically, the framework first predicts the average tuple processing time of a given scheduling solution based on the application graph and runtime statistics, then it assigns threads to machines under the guidance of prediction results. The authors of [16] proposed a dynamic optimization algorithm for Storm based on the theory on constraints (STDO-TOC), which dynamically eliminates the performance bottleneck of a topology. In addition, they proposed a real-time scheduling algorithm based on topology and traffic to effectively resolve the problem of inter-node load imbalance.

Recently, RL/DRL have been shown to be a useful technique for resource allocation and control of a big data and cloud computing systems. In [23], Naik *et al.*proposed a MapReduce scheduler in heterogeneous environments based on RL, which observes the system state of task execution and suggests speculative re-execution of the slower tasks to other available nodes in the cluster for faster execution. The authors of [26] propose a RL approach to automatically tune the configuration of MapReduce parameters, it has an initialization policy

with offline learning to reduce online learn time in different circumstances. In [20], Liu *et al.*proposed a hierarchical DRL-based framework to solve the resource allocation and power management problem in cloud computing systems. In a recent work, Li *et al.* [17] proposed to leverage DRL for scheduling in general-purpose DSDPSs, with the objective of minimizing average end-to-end tuple processing time.

Unlike these related works, we aim to develop an experience/data driven model-free approach for scheduling in DSDPSs to directly minimize the average tuple processing time using DRL with a variable number of threads, which has not been done before. Particularly, the control problems in MapReduce and cloud systems are quite different from our problem. Hence, those RL/DRL methods [20], [23], [26] cannot be applied here. Moreover, adaptive methods proposed in early works [36], [19] targeted at specific systems (such as continuous query system and multimedia streaming system); while we consider general-purposed DSDPSs, which have a quite different architecture. In addition, we aim to directly minimize the average tuple processing time, which is usually more effective than those methods [4], [38] that optimize indirect objectives (e.g., inter-node traffic load). Compared to related work [17] that considered the case where each PU has a fixed number of threads (specified by a user in advance), we propose to dynamically change the number of threads of each PU on the fly according to system states and demands, which makes the scheduling problem become much harder. Moreover, as mentioned above, we performed experiments under realistic settings (where multiple application instances are mixed up together), rather than a simplified single-instance setting used in most related works including [17].

V. Conclusions

In this paper, we present design, implementation and evaluation of a control framework, EXTRA, for scheduling in DSDPSs. EXTRA has two desirable features. First, it enables a DSDPS to dynamically change the number of threads during runtime according to system states and demands. Second, EXTRA leverages an experience/data driven model-free approach for dynamic control using the DRL, which enables a DSDPS to learn the best way to control itself from its own experience. We implemented EXTRA based on Apache Storm, and evaluated its performance with three representative SDP applications: continuous queries, word count (stream version), and log stream processing. Particularly, experiments were performed under realistic settings where multiple application instances are mixed up together. Extensive experimental results well justified effectiveness and robustness of EXTRA. Specifically, they show: 1) Compared to Storm's default scheduler and the state-of-the-art model-based method, EXTRA substantially reduces average tuple processing time by 39.6% and 21.6% respectively on average. 2) EXTRA does lead to more flexible and efficient control by enabling the use of a variable number of threads. 3) EXTRA is robust in a highly dynamic environment with significant workload change. Although DRL seems a promising technique for enabling experience-driven model-free control in DSDPSs, there are still many challenges to its real-world application. For example, training with less data is always a desired feature in large-scale DSDPSs. Many new techniques, like meta learning [11] and transfer learning [42], can help to further improve the training efficiency and adaptability of DRL via learning from existing knowledge, rather than directly learning from scratch in a new environment with different application topology or scale.

Acknowledgements

This work was supported by US National Science Foundation (NSF) grant 1704662. The information reported here does not reflect the position or the policy of the federal government.

References

[1] Alice's Adventures in Wonderland, *http://www.gutenberg.org/files/11/11-pdf.pdf*
[2] Q. Anderson, Storm real-time processing cookbook, *PACKT Publishing*, 2013.
[3] Apache Hadoop, *http://hadoop.apache.org/*
[4] L. Aniello, R. Baldoni and L. Querzoni, Adaptive online scheduling in Storm, *Proceedings of ACM DEBS'2013*.
[5] G. D. Arnold, R. Evans, H. v. Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris and B. Coppin, Deep reinforcement learning in large discrete action spaces, *arXiv: 1512.07679*, 2016.
[6] P. Bakkum and K. Skadron, Accelerating SQL database operations on a GPU with CUDA, *Proceedings of the 3rd Workshop on General-Purpose Computation on GPU (GPGPU'10)*, pp. 94–103.
[7] I. Bedini, S. Sakr, B. Theeten, A. Sala and Peter Cogan, Modeling performance of a parallel streaming engine: bridging theory and costs, *Proceedings of IEEE ICPE'2013*, pp. 173–184.
[8] P. Bellavista, A. Corradi, A. Reale and N. Ticca, Priority-based resource scheduling in distributed stream processing systems for big data applications, *Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing*, 2014, pp. 363–370.
[9] S. Boyd and L. Vandenberghe, Convex Optimization *Cambridge University Press*, 2004.
[10] F. Chen, M. Kodialam and T. V. Lakshman, Joint scheduling of processing and shuffle phases in MapReduce systems, *Proceedings of IEEE Infocom'2012*, pp. 1143–1151.
[11] C. Finn, P. Abbeel, and S. Levine, Model-agnostic meta-learning for fast adaptation of deep networks, *ICML'17*, pp. 1126–1135.
[12] Gurobi Optimizer, *http://www.gurobi.com/*
[13] Heron, *https://apache.github.io/incubator-heron/docs/concepts/architecture/*
[14] Q. Jiang and S. Chakravarthy, Scheduling strategies for a data stream management system, *Computer Science & Engineering, BNCOD*, 2004, pp. 16–30.
[15] T. Li, J. Tang and J. Xu, Performance modeling and predicitive scheduling for distributed stream data processing, *IEEE Transactions on Big Data*, Vol. 2, No. 4, 2016, pp. 353–364.
[16] C. Li, J. Zhang and Y. Luo, Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm, *Journal of Network and Computer Applications*, 2017(87), pp. 100–115.
[17] T. Li, Z. Xu, J. Tang and Y. Wang, Model-free control for distributed stream data processing using deep reinforcement learning, *Proceedings of the VLDB Endowment*, Vol. 11, No. 6, 2018, pp. 705–718.
[18] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, Continuous control with deep reinforcement learning, *Proceedings of ICLR'2016*.
[19] B. Liu, Y. Zhu, M. Jbantova, B. Momberger and E. A Rundensteiner, A dynamically adaptive distributed system for processing complex continuous queries, *Proceedings of PVLDB'2005*, pp. 1338–1341.
[20] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang and Y. Wang, A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning, *Proceedings of ICDCS'2017*, pp. 372–382.
[21] Logstash - Open Source Log Management, *http://logstash.net/*

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, Human-level control through deep reinforcement learning, *Nature*, Vol. 518, No. 7540, 2015, pp. 529–533.

[23] N. Naik, A. Negi and V. Sastry, Performance improvement of MapReduce framework in heterogeneous context using reinforcement learning, *Procedia Computer Science*, Vol.50, 2015, pp. 169–175.

[24] M. Nicola and M. Jarke, Performance modeling of distributed and replicated databases, *IEEE Transactions on Knowledge Discovery and Data Engineering*, 2000, Vol. 12, No. 4, pp. 645–672.

[25] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, R-storm: Resource-aware scheduling in storm, *Proceedings of the 16th Annual Middleware Conference*, 2015, pp. 149–161.

[26] C. Peng, C. Zhang, C. Peng and J. Man, A reinforcement learning approach to map reduce auto-configuration under networked environment, *International Journal of Security and Networks*, Vol. 12, No. 3, 2017, pp. 135–140.

[27] M. Restelli, Reinforcement learning - exploration vs exploitation, 2015, *http://home.deib.polimi.it/restelli/MyWebSite/pdf/rl5.pdf*

[28] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller, Deterministic policy gradient algorithms, *Proceedings of ICML'2014*.

[29] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctoc, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, Mastering the game of Go with deep neural networks and tree search *Nature*, 2016, pp. 484–489.

[30] Apache Spark, *http://spark.apache.org/*

[31] Apache Spark Streaming — Apache Spark, *http://spark.apache.org/streaming/*

[32] Active Streams, *https://www.cc.gatech.edu/systems/projects/AStreams/*

[33] Apache Storm, *http://storm.apache.org/*

[34] R. Sutton and A. Barto, Reinforcement learning: an introduction, *MIT press Cambridge*, 1998.

[35] TensorFlow, *https://www.tensorflow.org/*

[36] R. Tusch, Towards an adaptive distributed multimedia streaming server architecture based on service-oriented components, *Proceedings of JMLC'2003*, pp. 78–87.

[37] Y. Wei, V. Prasad, S. Son and J. Stankovic, Prediction-based QoS management for real-time data streams, *Proceedings of IEEE RTSS'2006*, pp. 344–358.

[38] J. Xu, Z. Chen, J. Tang and S. Su, T-Storm: traffic-aware online scheduling in Storm, *Proceedings of IEEE ICDCS'2014*, pp. 535–544.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, *Proceedings of OSDI'2008*, Vol. 8, No. 4, pp. 7.

[40] Apache Zookeeper, *https://zookeeper.apache.org/*

[41] Y. Zhu, Y. Jiang, W. Wu, L. Ding, A. Teredesai, D. Li and W. Lee, Minimizing makespan and total completion time in MapReduce-like systems, *Proceedings of IEEE Infocom'2014*, pp. 2166–2174.

[42] Z. Zhu, K. Lin, and J. Zhou, Transfer learning in deep reinforcement learning: A survey, *arXiv preprint*, 2020, arXiv:2009.07888.