May 2018

Efficient Online Scheduling in Distributed Stream Data Processing Systems

Teng Li
Syracuse University

Abstract

General-purpose Distributed Stream Data Processing Systems (DSDPSs) have attracted extensive attention from industry and academia in recent years. They are capable of processing unbounded big streams of continuous data in a distributed and real (or near-real) time manner. A fundamental problem in a DSDPS is the scheduling problem, i.e., assigning threads (carrying workload) to workers/machines with the objective of minimizing average end-to-end tuple processing time (or simply tuple processing time). A widely-used solution is to distribute workload over machines in the cluster in a round-robin manner, which is obviously not efficient due to the lack of consideration for communication delay among processes/machines. A scheduling solution makes a significant impact on the average tuple processing time. However, their relationship is very subtle and complicated. It does not even seem possible to have a mathematical programming formulation for the scheduling problem if its objective is to directly minimize the average tuple processing time.

In this dissertation, we first propose a model-based approach that accurately models the correlation between a scheduling solution and its objective value (i.e. average tuple processing time) for a given scheduling solution according to the topology of the application graph and runtime statistics. A predictive scheduling algorithm is then presented, which assigns tasks (threads) to machines under the guidance of the proposed model. This approach achieves an average of 24.9% improvement over Storm's default scheduler. However, the model-based approach still has its limitations: the model may not be able to fully capture the features of a DSDPS; prediction may not be accurate enough; and a large amount of high-dimensional data may lead to high overhead.

To address the limitations, we develop a model-free approach that can learn to control a DSDPS from its experience rather than adopting accurate and mathematically solvable

system models, just as a human learns a skill (such as cooking, driving, swimming, etc.). Recent breakthrough of Deep Reinforcement Learning (DRL) provides a promising approach for enabling effective model-free control. The proposed DRL-based model-free approach minimizes the average end-to-end tuple processing time by jointly learning the system environment via collecting very limited runtime statistics and making decisions under the guidance of powerful Deep Neural Networks (DNNs). This approach achieves great performance improvement over the current practice and the state-of-the-art model-based approach.

Moreover, there is still room for improvement for the above model-free approach: For the above model-free approach and most existing methods, a user specifies the number of threads for an application in advance without knowing much about runtime needs, which, however, remains unchanged during runtime. This could severely affect the performance of a DSDPS. Therefore, we further develop another model-free approach using DRL, EXTRA, which enables the dynamic use of a variable number of threads at runtime. It has been shown by extensive experimental results, by adding this new feature, EXTRA can achieve further performance improvement and greater flexibility on scheduling.

EFFICIENT ONLINE SCHEDULING IN DISTRIBUTED STREAM DATA PROCESSING SYSTEMS

By

Teng Li
B.S. (Applied Mathematics), Beijing University of Posts and Telecommunications, 2010

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Computer Engineering

Syracuse University
May 2018

ACKNOWLEDGMENTS

I would like to express my sincere appreciation for all the guidance, supports and company I had received during my rewarding Ph.D. journey.

I would like to express my deepest gratitude to my advisor, Dr. Jian Tang. for the continuous support of my Ph.D. study and research. Your patience, motivation and immense knowledge sets me a living example of a great researcher and a mentor. Your guidance is a beacon leading me through the research journey full of thorns, helping me get thorough the darkest days. You point out my weakness and help me improve. I could not have imagined having a better advisor for my Ph.D. study.

Besides my advisor, I would like to thank the committee members for all your insightful feedbacks and encouragements.

I also would like to thank my fellow lab mates for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the past six years. I wish you all the best.

My final but everlasting gratitude goes to my family, who have been a constant source of love and strength. Your sacrificial love and prayers have strengthened me to finish my study through the toughest times. I owe you my deepest respect and gratitude.

Table of Contents

Acknowledgments **v**

1 Introduction **1**

1.1 Motivations 1

1.2 Background 5

1.2.1 Distributed Stream Data Processing System (DSDPS) 5

1.2.2 Apache Storm 8

1.2.3 Deep Reinforcement Learning (DRL) 8

1.3 State of the Art and Literature Gap 11

1.4 Contributions and Dissertation Organization 16

2 Model-based Scheduling for Distributed Stream Data Processing Systems (DS-DPSs) **18**

2.1 Overview 18

2.2 Design and Implementation of the Proposed Framework 19

2.2.1 Overview of Our Design 19

2.2.2 Performance Prediction-based Scheduling 21

2.2.3 Implementation Details 28

2.3 Performance Evaluation 31

2.3.1 Experimental Setup 31

2.3.2 Experimental Results and Analysis 34

2.4 Summary 39

3 Model-free Scheduling for DSDPSs using Deep Reinforcement Learning (DRL) 41

3.1 Overview 41

3.2 Design and Implementation of the Proposed Framework 42

3.2.1 Overview of Our Design 42

3.2.2 DRL-based Scheduling 44

3.2.3 Implementation Details 52

3.3 Performance Evaluation 53

3.3.1 Experimental Setup 53

3.3.2 Experimental Results and Analysis 54

3.4 Summary 62

4 Model-free Scheduling for DSDPSs with a Variable Number of Threads 64

4.1 Overview 64

4.2 Design and Implementation of the Proposed Framework 65

4.2.1 Overview of Our Design 65

4.2.2 DRL-based Scheduling with a Variable Number of Threads 66

4.2.3 Implementation Details 73

4.3 Performance Evaluation 74

4.3.1 Experimental Setup 74

4.3.2 Experimental Results and Analysis 77

4.4 Summary 82

5 Conclusion and Future Work 83

References 85

CHAPTER 1
INTRODUCTION

1.1 Motivations

Emerging big stream data applications demand the underlying system to process heterogeneous, fast and high-volume data in a distributed and timely manner. A widely-used solution is parallel/distributed data processing, which divides input data into many small pieces and processes them in parallel over a large number of physical machines. MapReduce [1] and Hadoop [2] have emerged as the *de facto* programming model and system for big data processing respectively. However, they are not suitable for continuous stream data or online processing because they were designed for offline batch processing of static data, in which all input data need to be stored on a distributed file system in advance.

Here we focus on general-purpose *Distributed Stream Data Processing Systems (DSDPSs)* (such as Apache Storm [3] and Google's MillWheel [6]), which deal with processing of unbounded streams of continuous data at scale distributedly in real or near-real time. Their programming models and runtime systems are quite different from those of MapReduce-based batch processing systems, such as Hadoop [2] and Spark [27], which usually handle static big data in an offline manner, and they employ different message passing, fault tolerance and resource allocation mechanisms that are designed specifically for online stream data processing.

In a Distributed Stream Data Processing System (DSDPS), an application is usually modeled using a directed graph, in which each vertex corresponds to a data source or a Processing Unit (PU), and edges indicate data flow.

A stream data processing system handles unbounded streams of data tuples, which may last for a long time, therefore, the tuple processing time, i.e., the time between when the tuple comes out of the data source and the time when its processing is completed and acknowledged, is usually used as the primary performance metric. A tuple may traverse multiple PUs and experience various latencies (such as processing latency at a PU and transfer latency between PUs) during its lifetime.

A fundamental problem in a DSDPS is the scheduling problem (i.e., assigning workload to workers/machines) with the objective of minimizing average (end-to-end) tuple processing time (i.e. end-to-end delay). A widely-used solution is to distribute workload over machines in the cluster in a round-robin manner [3], which is obviously not efficient due to lack of consideration for communication delay among processes/machines.

In distributed batch processing systems (such as MapReduce-based systems), an individual task's completion time can be well estimated [46], and the scheduling problem can be well formulated into a Mixed Integer Linear Programming (MILP) problem with the objective of minimizing the makespan of a job [47, 48]. Then it can be tackled by optimally solving the MILP problem or using a fast polynomial-time approximation/heuristic algorithm. However, in a DSDPS, a task or an application never ends unless it is terminated by its user. A scheduling solution makes a significant impact on the average tuple processing time. But their relationship is very subtle and complicated. It does not even seem possible to have a mathematical programming formulation for the scheduling problem if its objective is to directly minimize average tuple processing time.

We may be able to better solve the scheduling problem if we can accurately model the correlation between a scheduling solution and its objective value (i.e. average tuple processing time), that is, to predict/estimate average tuple processing time for a given scheduling solution. However, how to accurately model and predict tuple processing time in a DSDPS is quite challenging and has not yet been well studied.

Queueing theory has been applied for modeling and prediction in distributed database systems [5]. However, it does not work for general-purpose stream data processing due to the following reasons: 1) Queueing theory can only offer good estimation for queueing delay under a few assumptions (e.g, tuple arrivals follow a Poisson distribution, etc), which, may not hold in a complex distributed data processing system. 2) In queueing theory, many problems in a queueing network (instead of a single queue) remain unknown, while a distributed data processing system represents a fairly complicated multi-point to multi-point queueing network where tuples from a queue may be distributed to multiple downstream queues, and a queue may receive tuples from multiple different upstream queues.

We plan to propose a novel predictive scheduling framework to enable fast and distributed stream processing, which features application-aware modeling for performance prediction and predictive scheduling. The application-aware modeling will accurately model and predict the average tuple processing time for a given scheduling solution, according to the topology of the application graph and runtime statistics. The scheduling will assign tasks (threads) to machines under the guidance of the prediction results.

Even if we could develop a relatively accurate model to calculate the objective value win the knowledge of a given scheduling solution, the approach still suffers from two problems: 1) In a very complicated distributed computing environment such as a DSDPS, tuple processing time may be caused by many factors, which are not fully captured by the model. 2) Prediction for each individual component may not be accurate enough. 3) A large amount of high-dimensional statistics data need to be collected to build and update the model, which leads to high overhead.

Therefore it is desirable to develop a model-free approach that can learn to well control a DSDPS from its experience rather than adopting accurate and mathematically solvable system models, just as a human learns a skill (such as cooking, driving, swimming, etc.) Besides, recent breakthrough of Deep Reinforcement Learning (DRL) [41] provides a promising technique for enabling effective model-free control. DRL [41] (originally developed by a startup named DeepMind) enables computers to learn to play games, including Atari 2600 video games and one of the most complicated games, Go (AlphaGo [42]), and beat the best human players.

DRL consists of an offline Deep Neural Networks (DNN) construction phase, which correlates the value function with corresponding states and actions, and an online deep Q-learning phase for action selection, system control, and DNN updating. We believe DRL is especially promising for control in DSDPSs because: 1) It has advantages over other dynamic system control techniques such as model-based predictive control in that the former is model-free and does not rely on accurate and mathematically solvable system models (such as queueing models), thereby enhancing the applicability in complex systems with randomized behaviors. 2) it is capable of handling a sophisticated state space (such as AlphaGo [42]), which is more advantageous over traditional Reinforcement Learning (RL) [45]. 3) It is able to deal with time-variant environments such as varying system states and user demands.

However, direct application of the basic DRL technique, such as Deep Q Network (DQN) based DRL proposed in the pioneering work [41], may not work well here since it is only capable of handling control problems with a limited action space but the control problems (such as scheduling) in a DSDPS usually have a sophisticated state space, and a huge action (worker threads, worker processes, virtual/physical machines, and their combinations) space. Moreover, the existing DRL methods [41] usually need to collect big data (e.g., lots of images for the game-playing applications) for learning, which are additional overhead and burden for an online system. Our goal is to develop a method that only needs to collect very limited statistics data during runtime.

In summary, it is both rewarding and challenging to apply DRL techniques in a model-free approach to control DSDPSs.

Even if an approach described above could be developed, there is still room for improvement: while in those approaches (including most current state-of-the-art), a user specifies the number of threads/processes for a (logic) Processing Unit (PU) in an application in advance without knowing much about runtime needs, which, however, remains unchanged during runtime. This could severely affect the performance of the DSDPSs.

We aim to develop a scheduling framework for DSDPSs which is capable of integrating changing the number of threads into the data/experience-driven model-free control framework, which can fully embrace the power and flexibility brought by a variable number of threads. However, this new feature makes the scheduling problem much harder since the new problem involves jointly determining the number of threads and their assignment at the same time.

1.2 Background

In this section, we provide a brief background introduction to DSDPS, Apache Storm and DRL.

1.2.1 Distributed Stream Data Processing System (DSDPS)

The typical architecture of a Distributed Stream Data Processing System (DSDPS) is illustrated in Fig. 1.1. As mentioned above, a stream is defined as an unbounded sequence of tuples and an application is usually modeled as a directed graph (known as topology in Storm), which includes two types of components: data source (known as spout in Storm) and Processing Unit (PU, known as bolt in Storm). A data source reads data from external source(s) and then emits tuples into the topology. A PU consumes tuples from data sources or other PUs, and processes them using code provided by the user. A PU can either store data to a database, or pass it to some other units for further processing. These components are represented by vertices in the application graph, where directed edges indicate how data streams are routed. Data sources and PUs can be executed as many parallel tasks on multiple physical/virtual machines, therefore a task can be considered as an instance of a data source or processing unit. To ensure reliability, when the message ID of a tuple coming out of a data source successfully traverses the whole application graph, a special acker is called to inform the originating data source that message processing is complete. The tuple processing is the duration between when the data source emits the tuple and when it has been acked (fully processed). Note that we are only interested in the processing times of those tuples emitted by data sources since they reflect processing latency over the whole application. Clearly, a scheduling solution has a significant impact on the tuple processing time. There are 5 ways for grouping, which define how to route tuples between sets of tasks:

- Shuffle grouping: Tuples are randomly distributed across receiving bolt's tasks and each task is guaranteed to receive an equal number of tuples.
- Fields grouping: A field of a tuple is used as the key to partition the stream. Tuples with the same key will be mapped to the same task.
- All grouping: Each tuple is broadcasted to all tasks of the corresponding bolt.
- Global grouping: The entire stream is routed to one of the bolt's tasks, usually the task with the lowest ID.
- Direct grouping: The producer of the stream decides which task of the consuming bolt will receive each tuple.

A DSDPS usually uses two levels of abstractions (logical and physical) to express parallelism. In the physical layer, it usually consists of a master (known as Nimbus in Storm) that serves as the central control unit, and a set of machines (called worker nodes in Storm) that actually process incoming data. An application graph (topology) is executed on multiple worker processes (called workers in Storm) running on one or multiple worker nodes. Each worker node runs a daemon called supervisor that listens for any work assigned to it by the scheduler. Slots are configured on each worker node. The number of slots indicates the number of workers that can be run on this worker node, and is usually pre-configured by the cluster operator based on hardware constraints such as the number of CPU cores. Each worker uses multiple threads (known as executors in Storm) to actually process data according to user code.

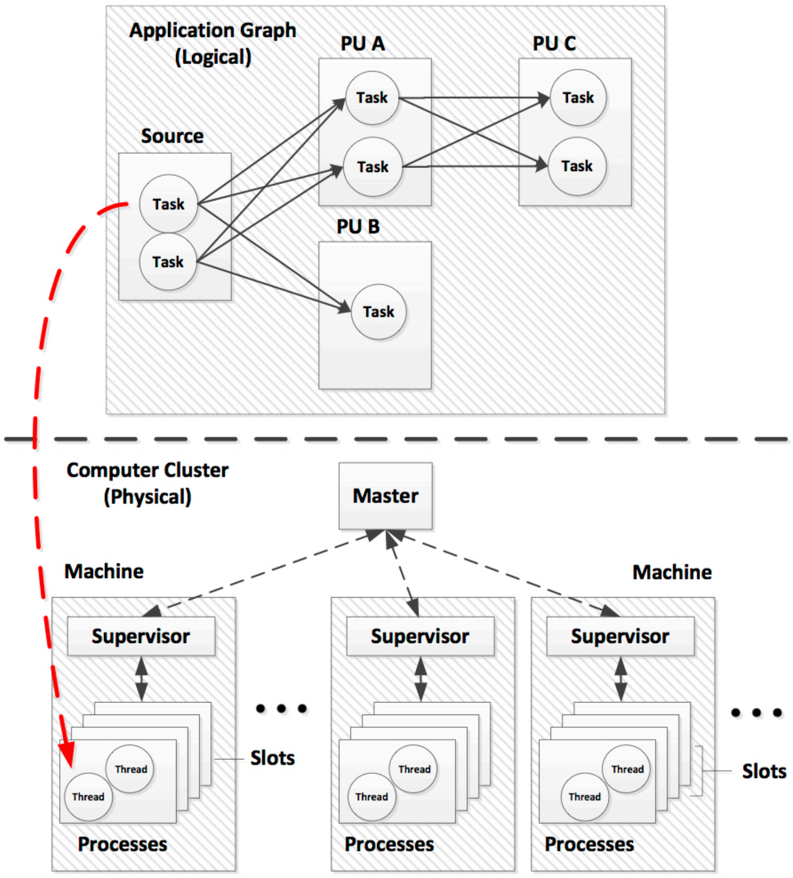


Fig. 1.1: The typical architecture of a Distributed Stream Data Processing System (DS-DPS)

The master is responsible for distributing users' code around the cluster, scheduling tasks, and monitoring them for failures. A *scheduling solution* specifies how to assign threads (executors) to processes (workers) and physical/virtual machines (worker nodes).

Storm includes a default scheduler, which uses a simple scheduling solution, which assigns executors to pre-configured workers in a round-robin manner and then evenly assigns those workers to available slots on worker nodes. This scheduling solution leads to almost even distribution of executors over available slots.

1.2.2 Apache Storm

Since we implemented the proposed frameworks based on Apache Storm [3], we briefly introduce it here. Apache Storm is an open-source and fault-tolerant DSDPS, which has an architecture and programming model very similar to what described above, and has been widely used by quite a few companies and institutes. In Storm, data source, PU, application graph, master, worker process and worker thread are called spout, bolt, topology, Nimbus, worker and executor, respectively. Storm uses *ZooKeeper* [43] as a coordination service to maintain it's own mutable configuration (such as scheduling solution), naming, and distributed synchronization among machines. All configurations stored in ZooKeeper are organized in a tree structure. Nimbus (i.e., master) provides interfaces to fetch or update Storm's mutable configurations. A Storm topology contains a topology specific configuration, which is loaded before the topology starts and does not change during runtime.

1.2.3 Deep Reinforcement Learning (DRL)

A standard RL setup consists of an agent interacting with an environment in discrete decision epochs. The agent observes state $\mathbf{s}_t$, takes an action $\mathbf{a}_t$ and receives a reward r_t at each decision epoch t. The goal is to find a policy $\pi(\mathbf{s})$ that maps a state to an action or actions deterministically or probabilistically to maximize the discounted cumulative reward $R_0 = \sum_{t=0}^{T} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)$, where $r(\cdot)$ is the reward function and $\gamma \in [0, 1]$ is the discount factor.

DRL [41] extends the well-known Q-learning to support end-to-end system control based on high-dimensional sensory input (such as raw images) using a Deep Neural Network (DNN). During the training phase, a DNN called Deep Q-Network (DQN) is built to derive the correlation between each state-action pair $(\mathbf{s}_t, \mathbf{a}_t)$ of the system under control and its value function $Q(\mathbf{s}_t, \mathbf{a}_t)$, which is the expected discounted cumulative reward. if the system is in state $\mathbf{s}_t$ and follows action $\mathbf{a}_t$ at decision epoch t (and a certain policy π thereafter):

$$Q(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}\Big[R_t | \mathbf{s}_t, \mathbf{a}_t\Big], \tag{1.1}$$

where $R_t = \sum_{k=t}^{T} \gamma^k r(\mathbf{s}_t, \mathbf{a}_t)$. A commonly-used off-policy method takes the greedy policy: $\pi(\mathbf{s}_t) = \operatorname{argmax}_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t)$. The DQN can be trained by minimizing the loss:

$$L(\boldsymbol{\theta}^Q) = \mathbb{E}\Big[y_t - Q(\mathbf{s}_t, \mathbf{a}_t | \boldsymbol{\theta}^Q)\Big], \tag{1.2}$$

where $\boldsymbol{\theta}^Q$ is the weight vector of the DQN and y_t is the target value, which can be estimated by:

$$y_t = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma Q(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1} | \boldsymbol{\theta}^{\boldsymbol{\pi}}) | \boldsymbol{\theta}^Q). \tag{1.3}$$

Using a neural network (or even a DNN) as the function approximator in RL is not new. However, such a non-linear approximator is known to suffer from instability or even divergence. Two effective techniques were introduced in [41] to improve stability: experience replay and target network. Unlike traditional RL, a DRL agent updates the DNN with a mini-batch from an experience replay buffer [41], which stores state transition samples collected during learning. Compared to the original Q-learning using only immediately collected samples, DRL randomly sample from the experience replay buffer. This allows a DRL agent to break the correlation among sequentially generated samples, and learn from a more independently and identically distributed past experiences, which is required by most of the training algorithms, such as Stochastic Gradient Descent (SGD). So experience replay can smooth out learning as well as avoiding oscillations or divergence. In addition, a DRL agent introduces a separate target network (with the same structure as DQN) to estimate target values $< y_t >$ for training the DQN, whose parameters, however, are slowly updated with the DQN weights every $C > 1$ epochs and are held fixed between individual updates.

The DQN-based DRL only works for control problems with a low-dimensional discrete action space. It is not trivial to extend it to deal with a continuous or high-dimensional discrete action space since it needs to find the action that maximizes the action-value function, which, however, requires an iterative process to solve a non-trivial non-linear optimization problem (continuous) or search for the best action in a large space at each epoch.

The commonly-used method for continuous control is the actor-critic approach [49], which employs the policy gradient to search for the optimal policy. The traditional actor-critic approach can also be extended to embrace DNN (such as DQN) to guide decision making [59]. For example, a recent work [59] from DeepMind introduced a deterministic actor-critic method, called Deep Deterministic Policy Gradient (DDPG), which has been shown to be very effective for continuous control. The basic idea is to maintain a parameterized actor function $\pi(\mathbf{s}_t|\boldsymbol{\theta}^{\pi})$ and a parameterized critic function $Q(\mathbf{s}_t, \mathbf{a}_t|\boldsymbol{\theta}^{Q})$. The critic function can be implemented using the above DQN, which returns Q value for a given state-action pair. The actor function can also be implemented using a DNN, which specifies the current policy by mapping a state to a specific action. According to [54], the actor network can be updated by applying the chain rule to the expected cumulative reward J with respect to the actor parameters $\boldsymbol{\theta}^{\pi}$:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}^{\pi}} J &\approx \mathbb{E}\Big[\nabla_{\boldsymbol{\theta}^{\pi}} Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\theta}^{Q})|_{\mathbf{s}=\mathbf{s}_t, \mathbf{a}=\pi(\mathbf{s}_t|\boldsymbol{\theta}^{\pi})}\Big] \\ &= \mathbb{E}\Big[\nabla_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\theta}^{Q})|_{\mathbf{s}=\mathbf{s}_t, \mathbf{a}=\pi(\mathbf{s}_t)} \cdot \nabla_{\boldsymbol{\theta}^{\pi}} \pi(\mathbf{s}|\boldsymbol{\theta}^{\pi})|_{\mathbf{s}=\mathbf{s}_t}\Big].\end{aligned} \tag{1.4}$$

Note that the experience replay and target network introduced above can also be integrated to this approach to ensure stability.

1.3 State of the Art and Literature Gap

Distributed/Parallel Stream Data Processing Systems (DSDPSs):

Recently, several general-purpose distributed/parallel systems have been developed particularly for stream data processing. As introduced above, Apache Storm [3] is an open-source, distributed, reliable and fault-tolerant DSDPS that is designed particularly for on-line processing of unbounded stream data. Storm provides a directed graph based model for programming as well as an effective mechanism to guarantee message processing. It supports fault-tolerance by monitoring worker processes and re-starting failed ones in a timely manner. Apache S4 (Simple Scalable Streaming System) [4] is another general-purpose, distributed, scalable and pluggable platform for processing continuous and unbounded streams of data. Time-Stream [7] is a distributed system designed by Microsoft specifically for low-latency continuous processing of big streaming data in the cloud. It employs a powerful new abstraction called resilient substitution that caters to the specific needs in this new computation model to handle failure recovery and dynamic reconfiguration in response to load changes. MillWheel [6] is a platform for building low-latency data processing applications, which is widely used at Google. In MillWheel, users specify a directed computation graph and application code for individual nodes, and the system manages persistent states and the continuous flows of data, all within the envelope of the framework's fault-tolerance guarantees. In addition, Apache Flink [18] is a open source framework for distributed big data analytics, the core of which is a distributed streaming data flow engine written in Java and Scala.

A few other distributed systems were developed to extend MapReduce to support stream data processing. In [20], Kumar *et al.* extended IBM's System S stream processing middleware with support for MapReduce by providing language and runtime support for specifying and embedding MapReduce jobs as elements of a larger data flow. In [21], a modified MapReduce architecture was introduced, which allows data to be pipelined between operators. This extends the MapReduce programming model beyond batch processing, and can reduce completion times and improve system utilization for batch processing jobs. Karve *et al.* in [22] proposed to use pipelining between the map and reduce phases to ensure the output of the map phase is made available to the reduce phase as soon as possible to speed up the execution of MapReduce jobs. In [23], Aly *et al.* presented a Hadoop-based system, M^3, which bypasses the HDFS to support main-memory-only processing and allows for continuous execution of the map and reduce phases where individual mappers and reducers never terminate. Backman *et al.* presented C-MR [24], a modified MapReduce processing model to continuously execute workflows of MapReduce jobs on unbounded data streams. Other MapReduce-like systems include Google's FlumeJava [25] and WalmartLab's Muppet [26], Twitter's Heron [55] and commercial solutions including Azure Stream Analytics [56] and Amazon Kinesis [57]. Muppet [26] was developed by WalmartLab for fast data processing, in which an update (instead of reduce) function is used to continuously update output based on data streams. FlumeJava was presented in [25], which is a Java library that makes it easy to develop, test, and run efficient data parallel pipelines. In addition, Apache Spark [27] is currently one of the most popular open source MapReduce-based frameworks. Spark Streaming [28] is a stream data processing system that uses the core Apache Spark API and its fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD transformations on those mini-batches of data. In [29], the authors presented design, implementation and evaluation of a novel GPU-accelerated system, G-Storm, for distributed stream data processing.

Unlike these related works that presented design and implementation of programming model and/or runtime systems for distributed stream data processing, in Chapter 2, our model-based scheduling framework focuses on performance modeling and prediction.

Modeling and Scheduling in DSDPSs: In [30], scheduling strategies aiming at minimizing tuple latency and total memory requirement were proposed for a Data Stream Management System (DSMS). The authors first proposed a strategy with a goal of minimizing tuple latency, followed by another one minimizing maximal memory requirement. This work mainly focused on deciding the processing order of the arriving tuples; while our work highlights the scheduling assignment of working threads. In another early work [31], the authors proposed novel operator scheduling approaches for data stream systems, which specify (1) which operators to schedule, (2) in which order to schedule the operators, and (3) how many tuples to process at each execution step. They studied the proposed approaches in the context of the Aurora data stream manager. In [32], an offline scheduler and an online scheduler were presented for Storm. The former works by analyzing the topology structure and making scheduling decisions; the latter works by continuously monitoring system performance and rescheduling executors at runtime to improve overall performance. Xu *et al.* presented T-Storm in [9], which focused on traffic-aware scheduling that minimizes inter-node and inter-process traffic in Storm while ensuring no worker nodes were overloaded. It also enables fine-grained control over worker node consolidation. In [33], Bellavista *et al.* proposed a general and simple technique to design and implement priority-based resource scheduling in flow-graph-based distributed stream processing systems by allowing application developers to augment flow graphs with priority metadata and by introducing an extensive set of priority schemas. In an early work [5], Nicola and Jarke provided a survey of performance models for distributed and replicated database systems, especially queueing theory based models. In [34], Wei *et al.* proposed a prediction based Quality-of-Service (QoS) management scheme for periodic queries over dynamic data streams. The scheme features novel query workload estimators which predict the query workload using execution time profiling and input data sampling. The authors of [35] described a decentralized framework for proactively predicting and alleviating hot-spots in distributed stream processing applications in real-time. In a recent work [36], the authors presented a novel technique for resource usage estimation of data stream processing workloads in the cloud, which uses mixture density networks, a combined structure of neural networks and mixture models, to estimate the whole spectrum of resource usage as probability density functions. In [37], Bedini *et al.* presented a set of models that formalize the performance characteristics of a practical distributed, parallel and fault-tolerant stream processing system using Actor Model theory. They modeled the characteristics of the data flow, the data processing and the system management costs at a fine granularity within the different steps of executing a distributed stream processing job. In addition, they presented an experimental validation of the proposed performance models using Storm. In a very recent work [50], Li *et al.* 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 solve the problem of inter-node load imbalance.

Unlike these related works, in Chapter 2, we present a application-aware method to predict average tuple processing time (rather than hotspot [35], workloads [36], or data processing and system management costs [37]) particularly for a typical big stream data processing system (such as Storm and S4), whose programming model and runtime system are different from those in [30, 31, 34]. In addition, we develop a predictive scheduling algorithm (that works closely with the proposed prediction method), which is different from priority-based methods [33] or those algorithms aiming at minimizing traffic between worker nodes [9, 32].

Moveover, unlike these related works, in Chapter 3 and Chapter 4, we also aim to develop an experience/data driven model-free approach for scheduling in DSDPSs to directly minimize the average tuple processing time with the application of the state-of-the-art DRL techniques, which has no been done before. Regarding dynamically changing the number of threads on the fly, Li *et al.* did include this feature in their work [50], however, the parameters were controlled by mathematical models, while ours takes a model-free approach with DRL techniques.

Deep Reinforcement Learning (DRL): DRL has recently attracted extensive attention from both industry and academia. On one hand, some works were focused on discrete control with a limited action space. In a pioneering work [41], Mnih *et al.* proposed Deep

Q Network (DQN), which can learn successful policies directly from high dimensional sensory inputs. This work bridges the gap between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging gaming tasks. The paper [60] considered a problem of multiple agents sensing and acting with the goal of maximizing their shared utility, based on DQN. The authors designed agents that can learn communication protocols to share information needed for accomplishing tasks. The authors of [61] proposed Double Q-learning as a specific adaptation to the DQN, which is introduced in a tabular setting and can be generalized to work with a large-scale function approximation. Schaul *et al.* proposed the prioritized experience replay in DQN in [51], in order to replay important transitions more frequently thus learn more efficiently. On the other hand, other works further extend DRL to address continuous control. Lillicrap *et al.* [59] proposed an actor-critic, model-free algorithm , DDPG, based on the deterministic policy gradient that can operate over continuous action spaces. Duan *et al.* [62] presented a benchmark suite of control tasks to quantify progress in the domain of continuous control. Gu *et al.* presented normalized advantage functions to reduce sample complexity for continuous tasks in [63]. Mnih *et al.* [53] presented asynchronous gradient descent for optimizing learning with DNNs with the result of successes of asynchronous actor-critic method on various continuous motor control tasks. Arnold *et al.* [64] extended the methods proposed for continuous actions to make decisions within a large discrete action space. In [58], the authors employed a DRL framework to jointly learn state representations and action policies using game rewards as feedback in the task of learning control policies for text-based games, where the action space contains all possible text descriptions. Gu *et al.* [52] proposed a policy gradient method named Q-Prop with the application of a Taylor expansion of the off-policy critic as a control variant.

In addition, RL/DRL has been applied to control of big data and cloud systems. In [71], the authors proposed a novel MapReduce scheduler in heterogeneous environments based on RL, which observes the system state of task execution and suggests speculative reexecution of the slower tasks on other available nodes in the cluster. An RL approach was proposed in [72] to enable automated tuning configuration of MapReduce parameters. Liu *et al.* [73] proposed a novel hierarchical framework for solving the overall resource resource allocation and power management problem in cloud computing systems with DRL. The control problems in MapReduce and cloud systems are quite different from the problem studied here. Hence, the methods proposed in these related works cannot be directly applied here.

Even though the emerging DRL has made tremendous successes on various control tasks, it remains unknown if and how it can be applied to solving complicated control and resource allocation problems in complex distributed computing systems such as DSDPSs. The proposed work makes one of the first efforts along this line.

1.4 Contributions and Dissertation Organization

In the dissertation, we propose efficient online scheduling in Distributed Stream Data Processing Systems (DSDPSs). Specifically, we make the following contributions:

- In Chapter 2, we propose a novel predictive scheduling framework to enable fast and distributed stream data processing, which features application-aware modeling for performance prediction and predictive scheduling. For prediction, we present a application-aware method to accurately predict the average tuple processing time of an application for a given scheduling solution, according to the topology of the application graph and runtime statistics. For scheduling, we present an effective algorithm to assign threads to machines under the guidance of prediction results.

- In Chapter 3, we present a novel model-free approach that can learn to well control a DSDPS from its experience rather than accurate and mathematically solvable system models, just as a human learns a skill (such as cooking, driving, swimming, etc). Specifically, we, for the first time, propose to leverage emerging Deep Reinforcement Learning (DRL) for enabling model-free control in DSDPSs; and present design, implementation and evaluation of a novel and highly effective DRL-based control framework, which minimizes average end-to-end tuple processing time by jointly learning the system environment via collecting very limited runtime statistics data and making decisions under the guidance of powerful Deep Neural Networks (DNNs).

- In Chapter 4, we present a model-free approach with a variable number of threads. The proposed approach enables a DSDPS to dynamically change the number of threads during runtime according to system states and demands. Moreover, the proposed approach 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.

The rest of the dissertation is organized as follows: We propose a novel predictive scheduling framework featuring application-aware modeling for performance prediction and predictive scheduling in Chapter 2. In Chapter 3, we propose a novel model-free approach that can learn to well control a DSDPS from its experience with the application of DRL. In Chapter 4, a model-free approach with a variable number of threads is proposed, enabling a DSDPS to dynamically change the number of threads on the fly during runtime, which extends the previous model-free approach and leads to more flexible and efficient control. The conclusion of the dissertation is presented in Chapter 5.

CHAPTER 2 MODEL-BASED SCHEDULING FOR DISTRIBUTED STREAM DATA PROCESSING SYSTEMS (DSDPSS)

2.1 Overview

In this chapter, we propose a novel predictive scheduling framework to enable fast and distributed stream data processing, which features topology-aware modeling for performance prediction and predictive scheduling. Moreover, we validate and evaluate the proposed framework based on an emerging platform, Apache Storm [3], which has been widely used in many companies and organizations, such as Twitter, Groupon, Yelp, etc. We summarize our contributions in the following:

- For prediction, we present a topology-aware method to accurately model and predict the average tuple processing time of an application for a given scheduling solution, according to the topology of the application graph and runtime statistics.
- For scheduling, we present an effective algorithm to assign tasks (threads) to ma-

chines under the guidance of the prediction results.

- We implemented the proposed framework based on Storm, and tested it with 3 representative applications: word count (stream version), log stream processing and continuous queries. It has potential to be applied to other similar platforms, such as Yahoo!'s S4 [4] Google's MillWheel [6] and Microsoft's TimeStream [7].
- Extensive experimental results well justify accuracy of the proposed prediction method and effectiveness of the proposed scheduling algorithm.

To the best of our knowledge, we are the first to propose a method for modeling and predicting the average tuple processing time in a distributed stream data processing system and validate the proposed method with the widely-used open-source platform, Storm.

2.2 Design and Implementation of the Proposed Framework

In this section, we present the design and implementation details of the proposed framework.

2.2.1 Overview of Our Design

As illustrated in Fig. 2.1, the proposed framework consists of several modules, including data collector, data store, time synchronizer, data pre-processor, performance predictor, schedule generator, and custom scheduler. The data collector runs as multiple threads to collect runtime statistics from all processes/machines in the cluster, which are then stored into the distributed data store. The data pre-processor prepares input data for prediction and scheduling based on raw data in the data store. The time synchronizer provides necessary synchronization for threads of the data collector.

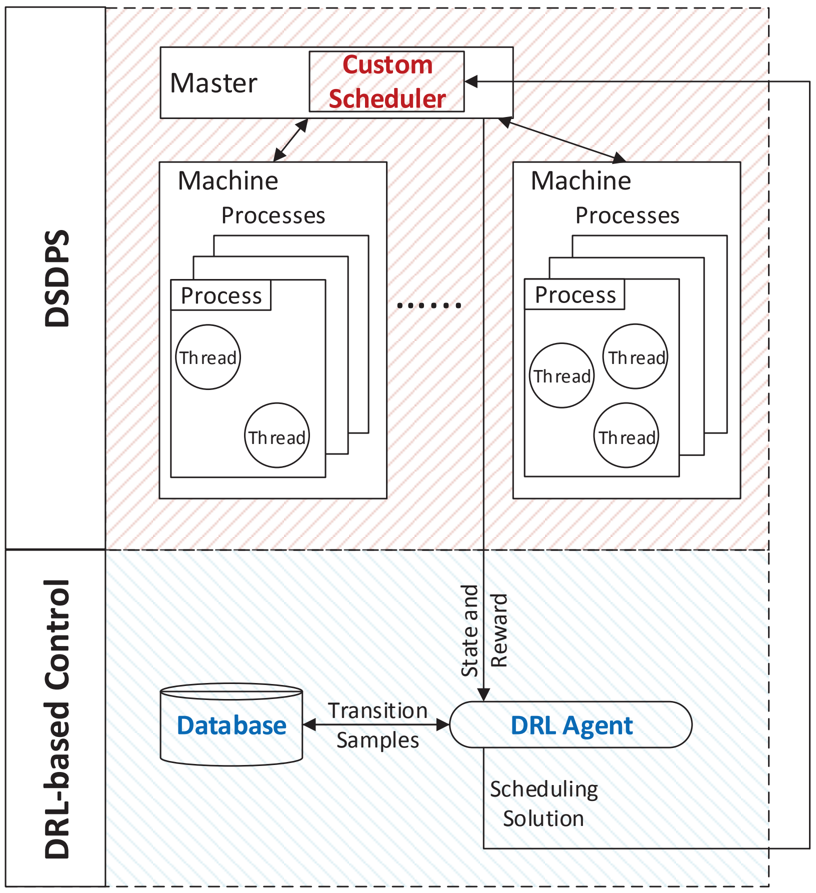


Fig. 2.1: The proposed predictive scheduling framework

The most important modules of the proposed framework include performance predictor, schedule generator and the custom scheduler. The performance predictor employs the proposed prediction method to predict the average tuple processing time of an application, which is introduced in Sections 2.2.2. The schedule generator generates a scheduling solution that defines how executors (i.e., instances of stream operators) are assigned to workers/worker nodes using the proposed scheduling algorithm (Section 2.2.2). The custom scheduler keeps reading the current scheduling solution from the schedule generator and deploys executors accordingly with the help of the master node on the fly.

Next, we introduce the proposed prediction and scheduling methods first, and then we describe the functions and implementation of these modules in details in Section 2.2.3. Note that the proposed modeling and prediction method works (or can be slightly modified to work) for any system in which an application is modeled using a directed graph and the proposed predictive scheduling algorithm works for any runtime system in which each PU is executed using multiple threads at runtime.

2.2.2 Performance Prediction-based Scheduling

Topology-aware Performance Prediction Model

We aim to model and predict the average tuple processing time of an application for a given scheduling solution, which can serve as a guideline to make a wise decision on task scheduling. Our idea is to predict the average tuple processing latency at each PU and the average tuple transfer latency between PUs (including both queueing and communication latencies) and add them up in a certain way according to the topology of the application graph. First, we show how to model the average tuple processing time of a typical application, whose topology is a chain. Then we extend it to graphs with general topologies.

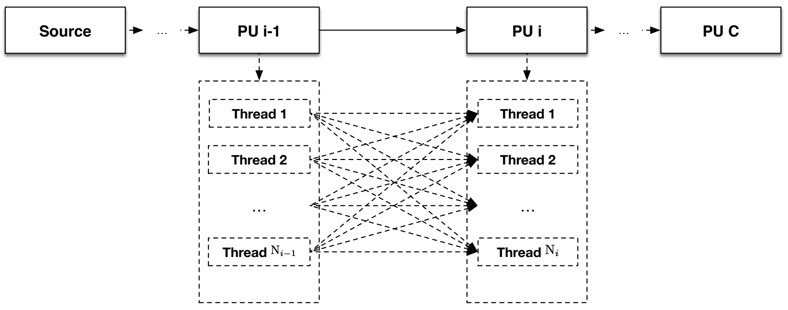


Fig. 2.2: A chain topology

First, we focus on a typical application graph with a chain topology as shown in Fig. 2.2. There are a total of $C + 1$ components in the topology with C PUs and a data source. Particularly, $i = 0$ corresponds to the data source. At runtime, the ith component ($i \in \{0, \cdots, C\}$) runs as N_i threads. Consider latency involving PU i ($i \in \{1, \cdots, C\}$), denoted as t_i^{PU}: after a tuple is processed by a thread of component (source or a PU) $(i - 1)$, it is inserted to an outgoing queue to a thread of PU i. After certain waiting time, it is passed to a thread of PU i, which is then inserted to an incoming queue in that thread. Again, after certain waiting time, it is processed by a thread of PU i. This latency can be obtained as follows:

$$t_i^{\text{PU}} = t_i^{\text{Proc}} + t_i^{\text{Tran}}, \tag{2.1}$$

where t_i^{Proc} and t_i^{Tran} are the average tuple processing latency at PU i and the average tuple transfer latency over edge $(i-1, i)$ respectively, which can be estimated using:

$$t_i^{\text{Proc}} = \sum_{j=1}^{N_i} w(i,j) * t_{ij}^{\text{Proc}}. \tag{2.2}$$

$$t_i^{\text{Tran}} = \sum_{j=1}^{N_{i-1}} \sum_{k=1}^{N_i} w(i,j,k) * t_{ijk}^{\text{Tran}}. \tag{2.3}$$

Here, t_{ij}^{Proc} is the average tuple processing latency of jth thread at PU i, which can be predicted using a supervised learning algorithm introduced later. $w(i,j)$ is the weight corresponding to that thread. In our implementation, it was set to the ratio of the number of tuples received at jth thread of PU i to the total number of tuples received by all threads of PU i within the past 10 minutes, which can be obtained using runtime statistics collected by the data collector. t_{ijk}^{Tran} is the average tuple transfer latency between the jth thread of PU $i-1$ and the kth thread of PU i, which includes communication delay and queueing delay on both ends as described above. This can also be predicted using a supervised learning algorithm introduced later. $w(i,j,k)$ is the corresponding weight, which was set to the ratio between the number of tuples sent from the jth thread of PU $i-1$ to the kth thread of PU i and the total number of tuples sent from PU $i-1$ to i within the past 10 minutes in our implementation. Note that our framework is not restricted to a particular weight function. Other weight functions may also be used here.

Let t_p denote the average tuple processing time of the chain topology, we have:

$$t_{\mathrm{p}} = \sum_{i=1}^{C} t_i^{\mathrm{PU}}. \tag{2.4}$$

Predicting the average tuple processing time of an application graph with a general topology is harder. However, an application graph is usually not very complicated so all paths from data sources can be easily enumerated. How to estimate average tuple processing time over *a path (i.e., a chain)* has been described above. Suppose that there are a total of P paths from a data source to the acker and the average tuple processing time of the pth path is t_p, then we have:

$$t_G = \max_{p \in \{1, \cdots, P\}} t_p. \tag{2.5}$$

Next, we use a simple example in Fig. 2.3 to illustrate how our performance model works on a general topology.

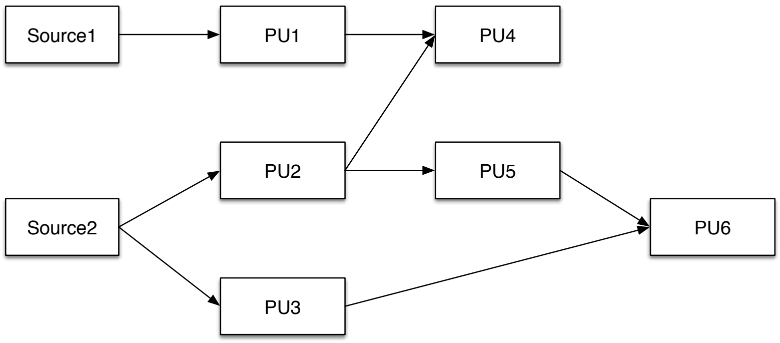


Fig. 2.3: An example for performance modeling

In this example, there are 2 data sources, 6 PUs and 2 ackers. There exist 4 paths: p1 = (Source1, PU1, PU4) and p2 = (Source2, PU2, PU4), p3 = (Source2, PU2, PU5, PU6), p4 = (Source2, PU3, PU6). Suppose at runtime, each of the data sources and PUs has 2 threads. In addition, shuffle grouping is used so that the work load is evenly distributed from an upstream thread to downstream threads. According to our model, the average tuple processing time is:

$$t_G = \max_{p \in \{1,\cdots,4\}} t_p. \tag{2.6}$$

Take path p3 = (Source2, PU2, PU5, PU6) as an example, the chain topology consists of 3 PUs and one source, thus $C = 3$. According to our assumption, $w(i,j) = 0.5, w(i,j,k) = 0.25, \forall i,j,k$. The average tuple processing time of path p3 can be estimated as:

$$\begin{aligned} t_3 &= \sum_{i=1}^{3} t_i^{\text{PU}} = \sum_{i=1}^{3} t_i^{\text{Proc}} + t_i^{\text{Tran}} \\ &= \sum_{i=1}^{3} (\sum_{j=1}^{2} 0.5 * t_{ij}^{\text{Proc}} + \sum_{j=1}^{2}\sum_{k=1}^{2} 0.25 * t_{ijk}^{\text{Tran}}). \end{aligned} \tag{2.7}$$

In order to use the above model for prediction, we need to predict the average tuple processing latency at a thread, t_{ij}^{Proc}, and the average tuple transfer latency between two threads, t_{ijk}^{Tran}. We propose to employ a supervised learning algorithm to predict the two types of latencies. Intuitively, they can be affected by many factors, including system configurations, workload of threads and machines, co-located threads, etc. We carefully select a set of features, which have been observed to have an impact on tuple processing and transfer latencies. We list the set of features selected for predicting the average tuple processing latency of a thread, t_{ij}^{Proc}, in the following:

- Machine-CPU: The number of CPU cores of the machine at which the target thread is located. This could be the total capacity of these CPU cores if machines have different CPUs.

- Machine-Memory: The size of memory (GB) of the machine at which the target thread is located.

- Thread-Workload: The number of tuples that the target thread processes within a window (30 seconds in the current implementation).

- Machine-Workload: The number of tuples that the machine (at which the target thread is located) processes within a window (30 seconds in the current implementation).

- Co-located-Threads: The total number of threads located at the same machine as the target thread.

Note that machines could be physical machines or Virtual Machines (VMs).

Predicting the average tuple transfer latency is more difficult since t_{jk}^{Tran} involves threads on both ends. We summarize our feature selection in the following:

- Machine-CPU-U: The number of CPU cores of the machine at which the upstream thread is located.

- Machine-Memory-U: The size of memory (GB) of the machine at which the upstream thread is located.

- Thread-Workload-U: The number of tuples that the upstream thread processes within a window (30 seconds in the current implementation).

- Machine-Workload-U: The number of tuples that the machine (at which the upstream thread is located) processes during a window (30 seconds in the current implementation).

- Machine-CPU-D: The number of CPU cores of the machine at which the downstream thread is located.

- Machine-Memory-D: The size of memory (GB) of the machine at which the downstream thread is located.

- Thread-Workload-D: The number of tuples that the downstream thread processes within a window (30 seconds in the current implementation).

- Machine-Workload-D: The number of tuples that the machine (at which the downstream thread is located) processes during a window (30 seconds in the current implementation).
- Co-location (boolean): if it is 1, the upstream thread and the downstream thread are located at the same machine; 0, otherwise.

Among all the selected features, some are general system features such as CPU and memory, which are well-known to have an impact on performance of a computing system. For example, in [38], Matsunaga *et al.* included CPU and memory of the machines as features in their training dataset. And, Xue *et al.* [39] chose them as features for their prediction framework. Other features are stream processing specific features such as workload and co-location, which have shown by our early work [9] to play an important role in system performance (i.e., average tuple processing time).

Since latency can be any real value, we need to use a regression algorithm to make prediction. This problem has high-dimensional input data with both continuous and categorical attributes, which makes it inappropriate to apply parametric regression algorithms such as linear and non-linear regression. The non-parametric regression algorithms, Support Vector Regression (SVR), is a good candidate for this problem because SVR scales relatively well to high dimensional data and the trade-off between classifier complexity and error can be controlled explicitly [8]. We used cross-validation for kernel function selection and the results show that radial basis kernel function outperforms the linear kernel, the polynomial kernel and the sigmoid kernel.

Predictive Scheduling

The prediction serves as a guideline for a schedule algorithm to assign threads to machines. Next, we discuss the scheduling problem and present the proposed algorithm.

Given the set of machines $\mathcal{M}$, the set of processes $\mathcal{Q}$, and the set of threads $\mathcal{N}$, the scheduling problem is to assign each thread to a process of a machine, that is, to find two mappings: $\mathcal{N} \mapsto \mathcal{Q}$ and $\mathcal{Q} \mapsto \mathcal{M}$. Our design ensures that on every machine, threads from the same application are assigned to no more than one process. This is enforced because assigning threads from one application to multiple (instead of one) processes on a machine introduces inter-process traffic and may lead to serious performance degradation, which has been shown in [9]. Therefore the two mappings can be merged to just one mapping: $\mathcal{N} \mapsto \mathcal{M}$, that is, to assign each thread to a machine.

The objective of the scheduling problem is to minimize the average tuple processing time over an application graph G. Let the scheduling solution be $X = \langle x_{ij} \rangle, i \in \{1, ..., N\}, j \in \{1, ..., M\}$, where $x_{ij} = 1$ if thread i is assigned to machine j. Note that different scheduling solutions can lead to different tuple processing and transfer latencies thus different tuple processing time.

The proposed algorithm computes a scheduling solution based on the average tuple processing time predicted by the method described above, which is denoted as Pred$(\cdot)$ in the following. We formally present the proposed predictive scheduling algorithm as Algorithm 1. The algorithm is essentially a greedy algorithm, which starts from an initial solution and keeps improving it by adjusting the thread-machine scheduling solution according to the prediction results. Then it tries to re-assign threads so as to achieve less average tuple processing time. The algorithm re-assigns one thread at a time in a greedy manner by picking the machine that minimizes Pred$(t_G(X))$.

In Algorithm 1, first we initialize X with a feasible solution X_0. In our implementation, X_0 was set to the solution given by Storm's default scheduler, which evenly distributes threads over all machines in an round robin manner. Then all threads are enumerated and traversed (Line 2), and for each thread, the algorithm tries to re-assign it to every possible machine (Line 3–7) that is different from the current solution to see if any improvement can be made. While it tries different solutions for threads, the prediction model produces the corresponding (predicted) average tuple processing time (Lines 8–9). Every time, the algorithm selects the solution that yields the smallest predicted value. Since the prediction

Algorithm 1 Predictive Scheduling Algorithm

Input: $\mathcal{M}, \mathcal{N}$
Output: $X = \langle x_{ij} \rangle, i \in \{1, ..., N\}, j \in \{1, ..., M\}$

```
 1: $X := X_0$, where $X_0$ can be any feasible solution;
 2: for $i = 1$ to $N$ do
 3:     for $j = 1$ to $M$ do
 4:         $X' := X$;
 5:         if $x_{ij} = 0$ then
 6:             $x'_{ij} := 1$;
 7:             $x'_{ik} := 0, \forall k \neq j$;
 8:             if Pred$(t_G(X'))$ < Pred$(t_G(X))$ then
 9:                 $X := X'$;
10:             end if
11:         end if
12:     end for
13: end for
14: return $X$;
```

model is already trained, Line 8 is considered to take constant time. Therefore the overall time complexity of Algorithm 1 is only $\mathcal{O}(MN)$.

2.2.3 Implementation Details

In this section, we describe the functions and implementation details of different modules in the proposed scheduling framework.

Data Collector

The data collector runs as multiple threads throughout all machines in the cluster. When a tuple arrives from another thread of a PU or a data source, the tuple transfer latency is recorded with a unique tuple ID. Then the tuple will be processed and sent to a thread of the next PU, while the tuple processing latency is again recorded by the data collector with the tuple ID. Timestamps are captured and inserted into data records and further sent to the distributed data store (described next). Every collected data record comes with a unique tuple ID, which ensures correct tracking of every tuple's processing and transfer latencies.

Data Store

The distributed data store works closely with the data collector. Whenever a data record is collected by the data collector, it will be sent to the distributed datastore, which is a shared data grid throughout the cluster. The data store is implemented by integrating Hazelcast [10] into the proposed framework, which is an open source in-memory data grid that performs well on distributed data storage and computation across machines and clusters. Usually, several data records will be generated by different threads at a time, which will be sent to the distributed data store simultaneously. In this case, these raw data records from different threads are mixed together disorderly and needs to be further processed by the data pre-processor (described next) so that they can be used for prediction.

Time Synchronizer

Threads of the data collector are distributed over processes/machines across the whole cluster, which require time synchronization over the network. The time synchronization is implemented by integrating PTP daemon (PTPd) [11] into the framework, which is an open source tool that implements the Precision Time Protocol (PTP) defined by the IEEE-1588 standard [12]. It makes available very precise time coordination of LAN connected computers, achieving clock accuracy in the order of sub-microsecond.

Data Pre-processor

Since the data records are collected from threads, processes and machines throughout the cluster, for every single tuple, there could be several corresponding data records in the data store, mixed with data records of other tuples. Those raw data records are shattered pieces and cannot be directly used for prediction. They need to be concatenated into new records such that there is a one-to-one correspondence between a data record and a tuple.

To achieve this, the data pre-processor congregates data records with the same tuple ID into a new record containing information needed for prediction. After the concatenation, every new record will consist of a tuple ID and corresponding processing and transfer latencies.

Performance Predictor

The performance predictor builds a prediction model using the method described in Section 2.2.2 based on the configuration information of the cluster and runtime data provided by the data pre-processor. Our framework is not restricted to the proposed prediction model. The other models can also be applied here.

Schedule Generator

The schedule generator generates a scheduling solution using the predictive scheduling algorithm described in Section 2.2.2 based on the prediction results provided by the performance predictor.

Custom Scheduler

After the scheduling solution is generated by the schedule generator, the custom scheduler deploys it on the cluster via the master. Note that our design handles schedule generation and deployment with two separate modules such that with our framework, the current scheduling algorithm can be replaced by a new one at runtime without shutting down the cluster.

Note that the overhead of the proposed predictive scheduling framework is not significant. First, traffic load for data collection is light since it only involves simple information such as timestamp and tuple ID. Second, data samples are first stored into local memory, which do not need to be collected frequently for prediction (every 20 minutes). Third, we use an efficient and mature software tool Hazelcast to create a separated channel for data collection, which further reduces its impact to regular tuple processing and communication. In addition, our framework starts to predict with limited data and then keeps updating the prediction model with new data. So it does not need to wait until a large amount of data are collected. Almost all the experimental results (presented later) show the original Storm and the Storm with the proposed framework stabilize after roughly the same amount of time (around 8 minutes), which justifies our argument.

2.3 Performance Evaluation

In this section, we describe experimental settings, followed by experimental results and analysis.

2.3.1 Experimental Setup

We implemented the proposed predictive scheduling framework over Apache Storm 0.9.2 release [13] (obtained from Storm's repository on Apache Software Foundation) and installed it on top of Ubuntu Linux 12.04. Experiments were performed on a cluster at Syracuse University's Green Data Center for performance evaluation. The Storm cluster consists of 10 IBM blade servers connected by a 1Gbps network, each of which was configured to have 10 slots. In order to fully test the performance of the proposed framework, 3 representative stream data processing applications (topologies) were implemented and deployed in our cluster: word count (stream version), log stream processing and continuous queries. We describe these topologies in the following.

Word Count Topology (stream version) (Fig. 2.4): Widely known as a MapReduce application, Word Count lists every word's number of appearances in a set of files. The stream version used in our experiments does a very similar job but with a stream data source. The topology has a chain-like structure with one spout and three bolts. The topology uses an open-source log agent called LogStash [14] to read data from input source files. LogStash submits input file lines as separate JSON values into a Redis [15] queue, which are consumed and emitted into the topology. The input file is the text file of Alice's Adventures in Wonderland [16]. The spout produces data stream while the input file is pushed into the queue. The spout is connected with the SplitSentence bolt which splits each input line into individual words and feeds them to a WordCount bolt that counts the number of appearances using fields grouping. The results are further sent to a Mongo bolt which stores them into a Mongo database. In the experiments, the topology was set to have 6 Spout executors (threads), 8 SplitSentence bolt executors, 8 WordCount bolt executors and 8 Mongo bolt executors.

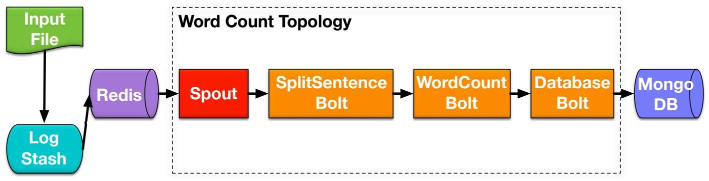


Fig. 2.4: Word Count Topology (stream version)

Word Count Topology (stream version) with 2 **sources**: This topology is almost same as the above expect that it includes an additional spout, which directly reads from the file [16]. In the experiment, the topology was set to have 6 executors for Spout 1, 6 executors for Spout 2, 8 SplitSentence bolt executors, 8 WordCount bolt executors and 8 Mongo bolt executors.

Log Stream Processing Topology (Fig. 2.5): Log stream processing represents one of the most popular use cases for Storm. The topology also uses Logstash [14] to read data from log files. LogStash submits log lines as separate JSON values into a Redis [15] queue which emits the output into the topology. Microsoft IIS log files obtained from the College of Engineering and Computer Science at Syracuse University were used as the input data. The LogRules bolt performs rule-based analysis on the log stream and delivers values containing a specific type of log entry instance. The log entry is then sent to two separate bolts simultaneously: one is the Indexer bolt which performs index actions and another one is the Counter bolt which performs counting actions on the log entries. For testing purpose, the original topology was slightly modified by introducing two Mongo bolts, one for the Indexer bolt and another one for the Counter bolt. These two Mongo bolts store the results into separate collections in a Mongo database for verification. In the experiments, the Storm cluster was configured to have 3 Spout executors (threads), 6 LogRules bolt executors, 6 Indexer bolt executors, 6 Count bolt executors and 6 executors for each Mongo bolt.

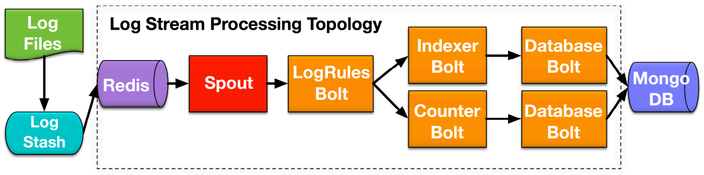


Fig. 2.5: Log Stream Processing Topology

Continuous Queries Topology (Fig. 2.6): This is another popular application on Storm. The topology is a select query that works by initializing access to a mapping table created in memory and looping over each row to check there is a hit [17].

It consists of a spout and two bolts. The spout continuously emits a randomly generated query, which is sent to a Query bolt. The database table is placed in memory on the host machine for Storm. The Query bolt takes queries from the spout and iterates over the database table to check if there are matching records. A hit on a matching record is then emitted to the last PU named the Printer bolt, which takes matching records and writes them to a file. In our experiments, we randomly generated a database table with vehicle plates and their owners' information (such as names and SSNs). Queries were also randomly generated to search the table for the speeding vehicles' owners (vehicle speed was randomly generated and attached to every entry). The experiments were conducted with 5 Spout executors (threads), 13 Query bolt executors and 13 Printer bolt executors.

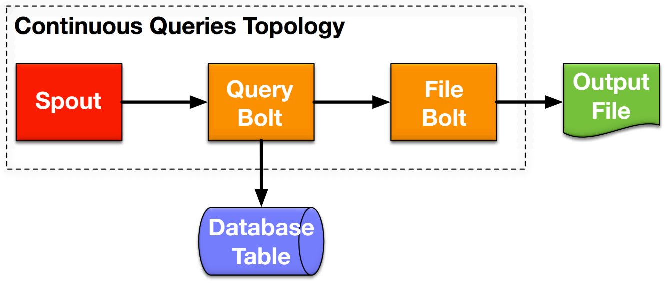
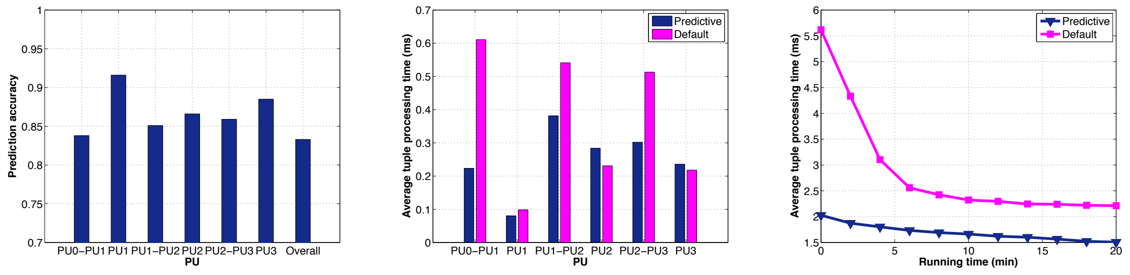
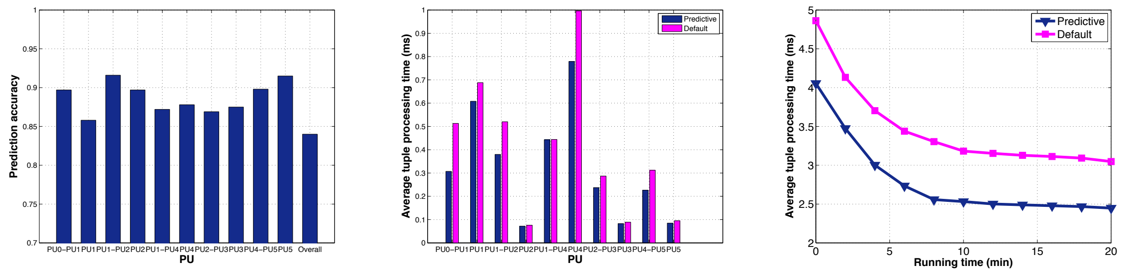


Fig. 2.6: Continuous Queries Topology



(a) Prediction accuracy (b) Average tuple processing/transfer latencies at/between PUs (c) Average tuple processing time (over the topology)

Fig. 2.7: Performance on a word count topology (stream version)

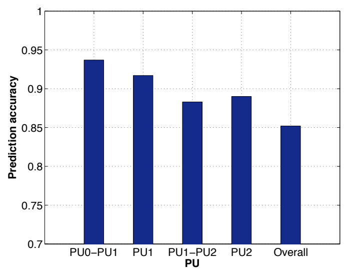


(a) Prediction accuracy (b) Average tuple processing/transfer latencies at/between PUs (c) Average tuple processing time (over the topology)

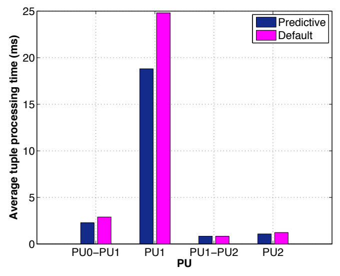
Fig. 2.8: Performance on a log stream processing topology

2.3.2 Experimental Results and Analysis

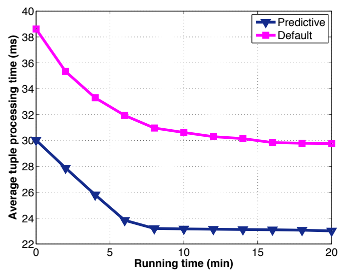
In this section, we present and analyze experimental results of the 3 applications introduced above. We used two commonly used metrics for performance evaluation, including predic-



(a) Prediction accuracy



(b) Average tuple processing/transfer latencies at/between PUs



(c) Average tuple processing time (over the topology)

Fig. 2.9: Performance on a continuous queries topology

tion accuracy and average tuple processing time (of a topology after stabilization) [9]. To justify the effectiveness of the proposed scheduling algorithm, Storm's default scheduler was used as the baseline solution for comparison.

For each topology, three figures are presented to show the experimental results. The first one shows the prediction accuracy, which is given by $(1 - \frac{|\text{Pred}-\text{Act}|}{\text{Act}})$, where Pred is the predicted value and Act is obtained from actual readings collected by data collector. For prediction, we used the readings of various features (described above) within the first 20 minutes after stabilization as the training data. During the training phase, randomly generated scheduling solutions were used to obtain sufficient amount of diverse data for training. In the corresponding figures, “PU0-PU1", “PU1" and “Overall" refer to the prediction accuracy of average (over 30 seconds) tuple transfer latency between PU0 and PU1, average tuple processing latency at PU1 and average tuple processing time of the whole topology after stabilization, respectively. The second figure shows the average (over 30 seconds) tuple processing time at a PU and average tuple transfer latency between PUs (after stabilization) given by the proposed scheduling framework and the default scheduler respectively. The third figure shows the average tuple processing time of the whole topology (given by the proposed scheduling framework and the default scheduler respectively) over a period of 20 minutes. From these experimental results, we can make the following observations.

Word Count Topology (stream version): The experimental results of this topology are shown in Fig. 2.7. From Fig. 2.7(a), we can see that the prediction accuracies for individual PUs vary from 83.8% (between the Spout and SplitSentence bolt) to 91.6% (at SplitSentence bolt). The prediction accuracy for the whole topology is 83.3%, which is slightly lower than individual prediction accuracies.

From Fig. 2.7(b), we can see that the predictive scheduling framework outperforms the default scheduler in terms of average tuple transfer latency. For example, the improvement on the average tuple transfer latency between PU0 and PU1 is the most significant one, which is over 60%. The improvements are around 30% for both PU1-PU2 and PU2-PU3. However, we also notice that the predictive scheduling framework performs slightly worse than the default scheduler at PU2 and PU3. This is because compared to the default scheduler (that leads to almost even distribution of workload), the predictive scheduling framework tends to consolidate workload to reduce tuple transfer latency by assigning more threads to a machine, which, however may lead to longer tuple processing latency at some PUs. Even though some PUs experience longer tuple processing latency, the predictive scheduling framework reduces the overall tuple processing time of the whole topology, which can also be observed from Fig. 2.7(c). From Fig. 2.7(c), we can observe that after a short initial period of about 8 minutes, the average tuple processing time (given by both schedulers) stabilizes at a lower value (compared to its initial value). The default scheduler's initial value is relatively high (5.6 ms) and then decreases significantly to about 2.2 ms. The predictive scheduling framework starts with a much lower initial value (2.0 ms) and stabilizes at about 1.5 ms, which represents a reduction of 31.8%.

When a topology is deployed, it will first be deployed on the master node (i.e., Nimbus in Storm). The master node will then distribute the topology file to worker nodes. Initially it takes some time for worker nodes to receive and deploy the topology file, and execute specified configurations. However, the measurement of tuple processing time starts from very beginning. Therefore, it can be observed that the average tuple processing time keeps decreasing and then stabilizes at certain value within the first few minutes. Similar observations can also be made in the following results.

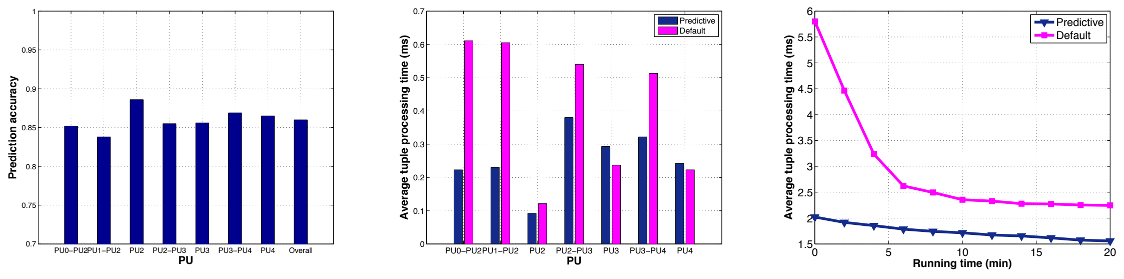
Log Stream Processing Topology: The experimental results of this topology are shown in Fig. 2.8. From Fig. 2.8(a), we can see that the prediction accuracies for individual PUs vary from 85.8% (at LogRules bolt) to 91.5% (between LogRules bolt and Indexer bolt). Similarly, the prediction accuracy for the whole topology is 84%, which is slightly lower than individual prediction accuracies.

From Fig. 2.8(b), we can discover that the predictive scheduling framework outperforms the default scheduler in terms of tuple transfer latency, while for different PUs, the improvement can vary, from 40% for PU0-PU1, to nearly flat for PU1-PU4. From Fig. 2.8(c), we can see that both schedulers stabilize after $8 - 10$ minutes. Two curves show similar trends. For the default scheduler, it starts with an initial value of 4.9 ms and stabilizes at about 3.0 ms. For the predictive scheduling framework, the initial value is lower (at 4.0 ms) and stabilizes at about 2.4 ms, which shows a 20% improvement.

Continuous Queries Topology: The experimental results of this topology are shown in Fig. 2.9. Fig. 2.9(a) shows prediction accuracies for different PUs vary from 88.3% (between Query bolt and Printer bolt) to 93.7% (between the Spout and Query bolt). For the whole topology, the prediction accuracy is 85.2%.

From Fig. 2.9(b), we can observe that the average tuple processing latency at PU1 (Query bolt) dominates the average tuple processing time of the topology due to lookup over a large size table. Still, the predictive scheduling framework achieves noticeable reduction on average tuple processing and transfer latencies. From Fig. 2.9(c), we can make similar observations as those from the previous two topologies. The average tuple processing time of the whole topology is longer than those of the two previous topologies due to heavier workload. The proposed framework still achieves a reduction of 22.7%, compared to the default scheduler.

Word Count Topology (stream version) with 2 sources: The experimental results of this topology are shown in Fig. 2.10. From Fig. 2.10(a), we can see that the prediction



(a) Prediction accuracy

(b) Average tuple processing/transfer latencies at/between PUs

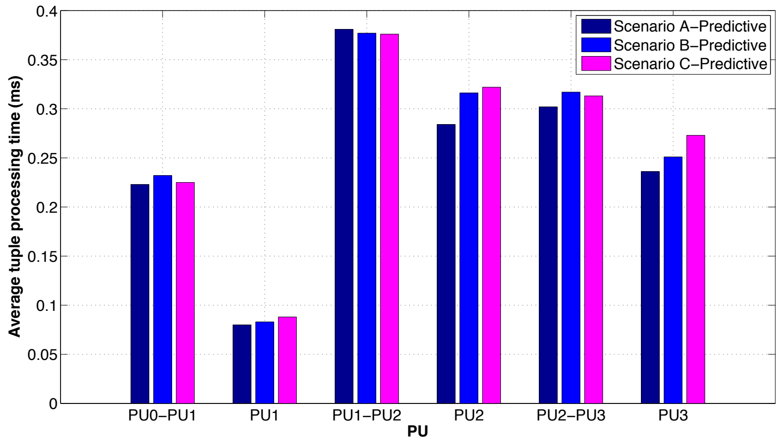
(c) Average tuple processing time (over the topology)

Fig. 2.10: Performance on a word count topology (stream version) with 2 sources

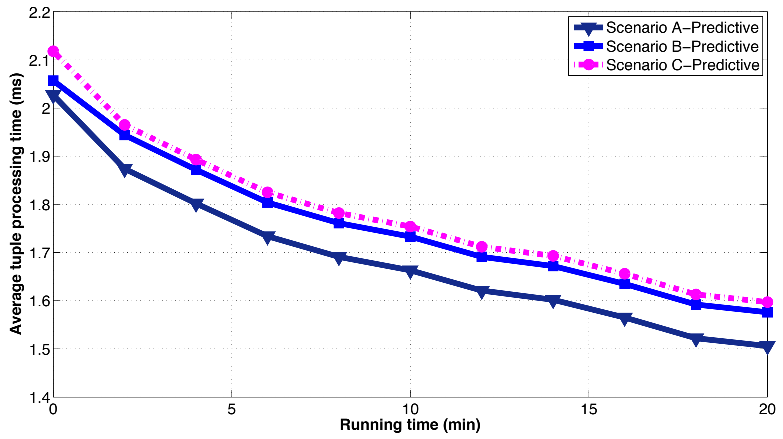
accuracies for individual PUs vary from 83.7% (between Spout 2 and SplitSentence bolt) to 88.2% (at SplitSentence bolt). The prediction accuracy for the whole topology is 86%. From Fig. 2.10(b), it can be seen that the performance behavior and improvement are similar to those in the single-source case. From Fig. 2.10(c), we can observe that the default scheduler leads to an initial value of about 5.8 ms and then decreases to around 2.24ms, while the predictive scheduling framework starts at about 2.0 ms and stabilizes at about 1.56 ms, which shows a reduction of 30.4%.

Performance Impact of Parameter Selection: In our experiments, one of the most important parameters is the number of executors of data sources and PUs. Here we used Word Count topology (stream version) again to illustrate the impact of parameter selection. The original setup (scenario A) for the experiment was 6 Spout executors (threads), 8 SplitSentence bolt executors, 8 WordCount bolt executors and 8 Mongo bolt executors. In scenario B, we decreased the numbers of executors to 3 Spout executors, 4 SplitSentence bolt executors, 4 WordCount bolt executors and 4 Mongo bolt executors. While in scenario C, we further decreased the numbers to 1 Spout executor, 2 SplitSentence bolt executors, 2 WordCount bolt executors and 2 Mongo bolt executors. The experiment results are shown in Fig. 2.11.

We can observe from Fig. 2.11(a) that at PU1, the processing latency increases from 0.08 ms (scenario A) to 0.083 ms (scenario B) and further to 0.088 ms (scenario C), while average tuple transfer latencies remain almost the same. Similar observations can be made for other PUs. Moreover, it can be seen from Fig. 2.11(b) that more executors help reduce average tuple processing time over the topology as expected.



(a) Average tuple processing/transfer latencies at/between PUs



(b) Average tuple processing time (over the topology)

Fig. 2.11: Performance impact of parameter selection on a word count topology (stream version)

2.4 Summary

In this chapter, we presented the design, implementation and evaluation of a model-based predictive scheduling framework aiming at fast and distributed stream data processing. The framework consists of several functional modules including data collector, data store, time synchronizer, data pre-processor, performance predictor, schedule generator and custom scheduler. Together they form a complete framework that can collect runtime data, analyze them, make predictions and come up with a scheduling solution. For prediction, a topology-aware model was presented to accurately predict the average tuple processing time of an application for a given scheduling solution, using runtime statistics. For scheduling, an effective algorithm was presented to assign threads to machines under the guidance of prediction results. We implemented it based on Apache Storm, and tested it in a cluster with 3 representative applications: word count (stream version), log stream processing and continuous queries. Extensive experimental results show 1) The topology-aware prediction method offers an average accuracy of 84.2%. 2) The predictive scheduling framework reduces the average tuple processing time by 24.9% on average, compared to the Storm's default scheduler.

Chapter 3
Model-free Scheduling for DSDPSs using Deep Reinforcement Learning (DRL)

3.1 Overview

In this chapter, we aim to develop a novel and highly effective DRL-based model-free control framework for a DSDPS to minimize average data processing time by jointly learning the system environment with very limited runtime statistics data and making scheduling decisions under the guidance of powerful *Deep Neural Networks (DNNs)*. We summarize our contributions in the following:

- We show that direct application of DQN-based DRL to scheduling in DSDPSs does not work well.
- We are the first to present a highly effective and practical DRL-based model-free control framework for scheduling in DSDPSs.
- We show via extensive experiments with three representative Stream Data Processing

(SDP) applications that the proposed framework outperforms the current practice and the state-of-the-art.

To the best of our knowledge, we are the first to leverage DRL for enabling model-free control in DSDPSs. We aim to promote a simple and practical model-free approach based on emerging DRL, which, we believe, can be easily extended to better control many other complex distributed computing systems.

3.2 Design and Implementation of the Proposed Framework

In this section, we present the design and implementation details of the proposed framework.

3.2.1 Overview of Our Design

We illustrate the proposed framework in Figure 3.1, which can be viewed to have two parts: DSDPS and DRL-based Control. The architecture is fairly simple and clean, which consists of the following components:

1) *DRL Agent (Section 3.2.2):* it is the core of the proposed framework, which takes the state as input, applies a DRL-based method to generating a scheduling solution, and pushes it to the custom scheduler.

2) *Database:* It stores transition samples including state, action and reward information for training (See Section 3.2.2 for details).

3) *Custom Scheduler:* It deploys the generated scheduling solution on the DSDPS via the master.

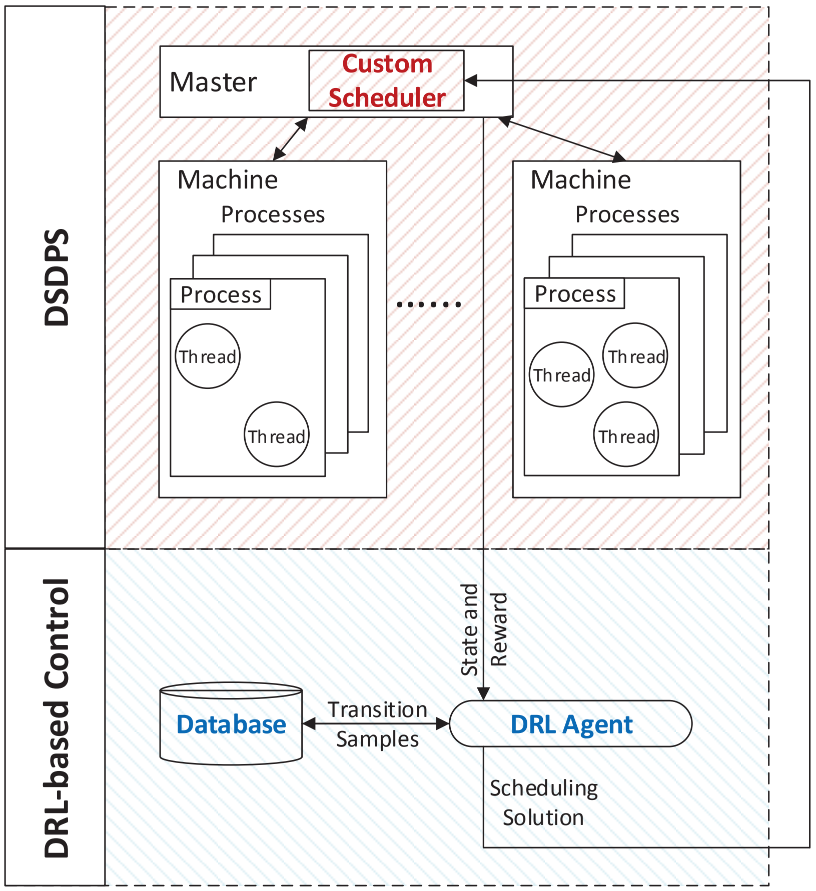


Fig. 3.1: The architecture of the proposed DRL-based control framework

Our design leads to the following desirable features:

1) *Model-free Control:* Our design employs a DRL-based method for control, which learns to control a DSDPS from runtime statistics data without relying on any mathematically solvable system model.

2) *Highly Effective Control:* The proposed DRL-based control is guided by DNNs, aiming to directly minimize average tuple processing time. Note that the current practice evenly distributes workload over machines; and some existing methods aim to achieve an indirect goal, (e.g., minimizing inter-machine traffic load [9]), with the hope that it can lead to minimum average tuple processing time. These solutions are obviously less convincing and effective than the proposed approach.

3) *Low Control Overhead:* The proposed framework only needs to collect very lim-

ited statistics data, i.e., just the average tuple processing time, during runtime for offline training and online learning (see explanations in Section 3.2.2), which leads to low control overhead.

4) *Hot Swapping of Control Algorithms:* The core component of the proposed framework, DRL agent, is external to the DSDPS, which ensures minimum modifications to the DSDPS, and more importantly, makes it possible to replace it or its algorithm at runtime without shutting down the DSDPS.

5) *Transparent to DSDSP users:* The proposed framework is completely transparent to DSDSP users, i.e., a user does not have to make any change to his/her code in order to run his/her application on the new DSDPS with the proposed framework.

3.2.2 DRL-based Scheduling

In this section, we present the proposed DRL-based control, which targets at minimizing the end-to-end average tuple processing time via scheduling.

Given a set of machines $\mathcal{M}$, a set of processes $\mathcal{P}$, and a set of threads $\mathcal{N}$, a scheduling problem in a DSDPS is to assign each thread to a process of a machine, i.e., to find two mappings: $\mathcal{N} \mapsto \mathcal{P}$ and $\mathcal{P} \mapsto \mathcal{M}$. It has been shown [9] that assigning threads from an application (which usually exchange data quite often) to more than one processes on a machine introduces inter-process traffic, which leads to serious performance degradation. Hence, similar as in [9, 66], our design ensures that on every machine, threads from the same application are assigned to only one process. Therefore the above two mappings can be merged into just one mapping: $\mathcal{N} \mapsto \mathcal{M}$, i.e., to assign each thread to a machine. Let a scheduling solution be $\mathbf{X} =< x_{ij} >, i \in \{1, \cdots, N\}, j \in \{1, \cdots, M\}$, where $x_{ij} = 1$ if thread i is assigned to machine j; and N and M are the numbers of threads and machines respectively. Different scheduling solutions lead to different tuple processing and transfer delays at/between tasks at runtime thus different end-to-end tuple processing times [66]. We aim to find a scheduling solution that minimizes the average end-to-end tuple processing time.

We have described how DRL basically works in Section 1.2.3. Here, we discuss how to apply DRL to solving the above scheduling problem in a DSDPS. We first define the state space, action space and reward.

State Space: A state $\mathbf{s} = (\mathbf{X}, \mathbf{w})$ consists of two parts: the scheduling solution $\mathbf{X}$, i.e., the current assignment of executors, and the workload $\mathbf{w}$, which is the tuple arrival rate (the number of tuples per second) of the application. The state space is denoted as $\mathcal{S}$. Workload is included in the state in order to achieve better adaptivity and sensitivity to the incoming workload, which has been validated by our experimental results.

Action Space: An action is defined as $\mathbf{a} =< a_{ij} >, \forall i \in \{1, ..., N\}, \forall j \in \{1, ..., M\}$, where $\sum_{j=1}^{M} a_{ij} = 1, \forall i$, and $a_{ij} = 1$ means assigning thread i to machine j. The action space $\mathcal{A}$ is the space that contains all feasible actions. Note that the constraints $\sum_{j=1}^{M} a_{ij} = 1, \forall i$ ensure that each thread i can only be assigned to a single machine, and the size of action space $|\mathcal{A}| = M^N$. Note that an action can be easily translated to its corresponding scheduling solution.

Reward: The reward is simply defined to be the negative average tuple processing time so that the objective of the DRL agent is to maximize the reward.

Note that the design of state space, action space and reward are critical to the success of a DRL method. In our case, the action space and reward are quite obvious. However, there are many different ways for defining the state space because a DSDPS includes various runtime information (features) [66], e.g., CPU/memory/network usages of machines, workload at each executor/process/ machine, average tuple processing delay at each executor/PU, tuple transfer delay between executors/PUs, etc. We, however, choose a simple and clean way in our design. We tried to add other system runtime information into the state but found that it failed to improve system performance. This is because different scheduling solutions lead to different values for the above features and eventually different average end-to-end tuple processing times, while the tuple arrival rate reflects the influence of incoming workload on the average end-to-end tuple processing time; and other information is redundant. We observed that based on our design, the proposed DNNs can well model the correlation between a scheduling solution (state) and average end-to-end tuple processing time (reward) after training. Designing the state space in this way can significantly reduce control overhead and relieve the system from the burden of heavy data collection because we only need to collect the average processing time at each decision epoch, which can be easily done in a DSDPS (e.g., using the Storm REST API).

A straightforward way to apply DRL to solving the scheduling problem is to directly use the DQN-based method proposed in the pioneering work [41]. The DQN-based method uses a value iteration approach, in which the value function $Q = Q(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta})$ is a parameterized function (with parameters $\boldsymbol{\theta}$) that takes state $\mathbf{s}$ and the action space $\mathcal{A}$ as input and return Q value for each action $\mathbf{a} \in \mathcal{A}$. Then we can use a greedy method to make an action selection:

$$\pi_Q(\mathbf{s}) = \operatorname*{argmax}_{\mathbf{a} \in \mathcal{A}} Q(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta}) \tag{3.1}$$

If we want to apply the DQN-based method here, we need to restrict the exponentially-large action space $\mathcal{A}$ described above to a polynomial-time searchable space. The most natural way to achieve this is to restrict each action to assigning only one thread to a machine. In this way, the size of the action space can be significantly reduced to $|\mathcal{A}| = N \times M$, which obviously can be searched in polynomial time. Specifically, as mentioned above, in the offline training phase, we can collect enough samples of rewards and the corresponding state-action pairs for constructing a sufficiently accurate DQN, using a model-free method that deploys a randomly-generated scheduling solution (state), and collect and record the corresponding average tuple processing time (reward). Then in the online learning phase, at each decision epoch t, the DRL agent obtains the estimated Q value from the DQN for each $\mathbf{a}_t$ with the input of current state $\mathbf{s}_t$. Then ϵ-greedy policy [44] is applied to select the action $\mathbf{a}_t$ according to the current state $\mathbf{s}_t$: with $(1 - \epsilon)$ probability, the action with the highest estimated Q value is chosen, or an action is randomly selected with probability ϵ. After observing immediate reward r_t and next state $\mathbf{s}_{t+1}$, a state transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ is stored into the experience replay buffer. At every decision epoch, the DQN is updated with a mini-batch of collected samples in the experience replay buffer using SGD.

Although this DQN-based method can provide solutions to the scheduling problem and does achieve model-free control for DSDPSs, it faces the following issue. On one hand, as described above, its time complexity grows linearly with $|\mathcal{A}|$, which demands an action space with a very limited size. On the other hand, restricting the action space may result in limited exploration of the entire exponentially-large action space and thus suboptimal or even poor solutions to the scheduling problem, especially for the large cases. The experimental results in Section 3.3 validate this claim.

The Actor-critic-based Method for Scheduling

In this section, we present a method that can better explore the action space while keeping time complexity at a reasonable level.

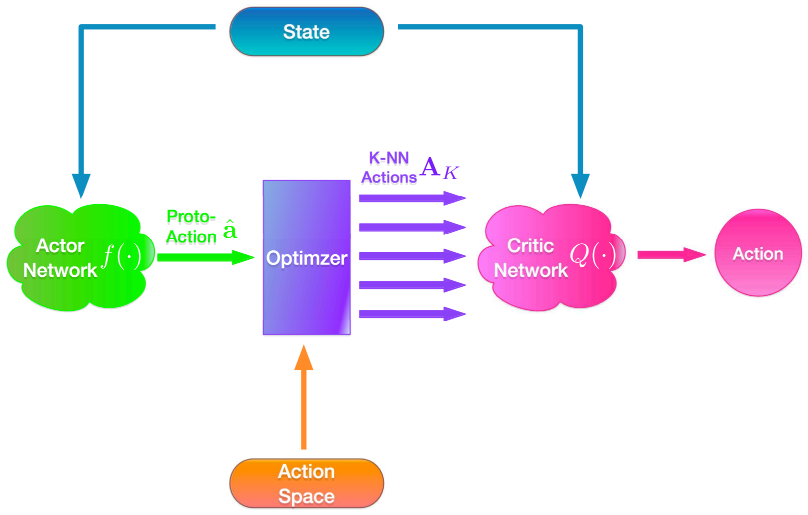

State
Actor Network $f(\cdot)$
Proto-Action $\hat{\mathbf{a}}$
Optimzer
K-NN Actions $\mathbf{A}_K$
Critic Network $Q(\cdot)$
Action
Action Space

Fig. 3.2: The actor-critic-based method

We leverage some advanced RL techniques, including actor-critic method [45, 64] and the deterministic policy gradient [54], for solving the scheduling problem. Note that since these techniques only provide general design frameworks, we still need to come up with a specific solution to our problem studied here. The basic idea of the proposed scheduling method is illustrated in Figure 3.2, which includes three major components: 1) an actor network that takes the state as input and returns a *proto-action* $\hat{\mathbf{a}}$, 2) an optimizer that finds a set $\mathbf{A}_K$ of K Nearest Neighbors (K-NN) of $\hat{\mathbf{a}}$ in the action space, and 3) a critic network that takes the state and and $\mathbf{A}_K$ as input and returns Q value for each action $\mathbf{a} \in \mathbf{A}_K$. Then an action with the highest Q value can be selected for execution. The basic design philosophy of this method is similar to that of a rounding algorithm, which finds a continuous solution by solving a relaxed version of the original integer (i.e., discrete) problem instance and then rounds the continuous solution to a “close" feasible integer solution that hopefully offers a objective value close to the optimal. Specifically, the actor network $f(\mathbf{s}; \boldsymbol{\theta}^{\boldsymbol{\pi}}) = \hat{\mathbf{a}}$ is a function parameterized by $\boldsymbol{\theta}^{\boldsymbol{\pi}}$ and $f : \mathcal{S} \mapsto \mathbb{R}^{MN}$. $\hat{\mathbf{a}}$ is returned as a proto-action that takes continuous values so $\hat{\mathbf{a}} \notin \mathcal{A}$. In our design, we use a 2-layer fully-connected feedforward neural network to serve as the actor network, which includes 64 and 32 neurons in the first and second layer respectively and uses the hyperbolic tangent function $\tanh(\cdot)$ for activation. Note that we chose this activation function because our empirical testing showed it works better than the other commonly-used activation functions.

The hardest part is to find the K-NN of the proto-action, which has not been well discussed in related work before. Even though finding K-NN can easily be done in linear time, the input size is M^N here, which could be a huge number even in a small cluster. Hence, enumerating all actions in $\mathcal{A}$ and doing a linear search to find the K-NN may take an exponentially long time. We introduce an optimizer, which finds the K-NN by solving a series of Mixed-Integer Quadratic Programming (MIQP) problems presented in the following:

MIQP-NN:

$$\begin{aligned}
\min_{\mathbf{a}}: \quad & \|\mathbf{a} - \hat{\mathbf{a}}\|_2^2 \\
\text{s.t.:} & \sum_{j=1}^{M} a_{ij} = 1, \forall i \in \{1, \cdots, N\}; \\
& a_{ij} \in \{0, 1\}, \forall i \in \{1, \cdots, N\}, \forall j \in \{1, \cdots, M\}.
\end{aligned} \tag{3.2}$$

In this formulation, the objective is to find the action **a** that is the nearest neighbor of the proto-action $\hat{\mathbf{a}}$. The constraints ensure that action **a** is a feasible action, i.e., $\mathbf{a} \in \mathcal{A}$. To find the K-NN, the MIQP-NN problem needs to be iteratively solved K times. Each time, one of the KNN of the proto-action will be returned, the corresponding values $< a_{ij} >$ are fixed, then the MIQP-NN problem is updated and solved again to obtain the next nearest neighbor until all the K-NN are obtained. Note that this simple MIQP problem can usually be efficiently solved by a solver as long as the input size is not too large. In our tests, we found our MIQP-NN problem instances were all solved very quickly (within 10ms on a regular desktop) by the Gurobi Optimizer [67]. For very large cases, the MIQP-NN problem can be relaxed to a convex programming problem [68] and a rounding algorithm can be used to obtain approximate solutions.

The set $\mathbf{A}_K$ of K-NN actions are further passed to the critic network to select the action. The critic network $Q(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta}^{\boldsymbol{Q}})$ is a function parameterized by $\boldsymbol{\theta}^{\boldsymbol{Q}}$, which returns Q value for each action $\mathbf{a} \in \mathbf{A}_K$, just like the above DQN. The action can then be selected as follows:

$$\pi_Q(\mathbf{s}) = \operatorname*{argmax}_{\mathbf{a} \in \mathbf{A}_K} Q(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta}^{\boldsymbol{Q}}). \tag{3.3}$$

Similar to the actor network $f(\cdot)$, we employ a 2-layer fully-connected feedforward neural network to serve as the critic network, which includes 64 and 32 neurons in the first and second layer respectively and uses the hyperbolic tangent function $\tanh(\cdot)$ for activation. Note that the two DNNs (one for actor network and one for critic network) are jointly trained using the collected samples.

Algorithm 2 The actor-critic-based method for scheduling

```
1: Randomly initialize critic network $Q(\cdot)$ and actor network $f(\cdot)$ with weights $\boldsymbol{\theta^Q}$ and $\boldsymbol{\theta^\pi}$ respectively;
2: Initialize target networks $Q'$ and $f'$ with weights $\boldsymbol{\theta^{Q'}} \leftarrow \boldsymbol{\theta^Q}$, $\boldsymbol{\theta^{\pi'}} \leftarrow \boldsymbol{\theta^\pi}$;
3: Initialize experience replay buffer **B**;
   /**Offline Training**/
4: Load the historical transition samples into **B**, train the actor and critic network offline;
   /**Online Learning**/
5: Initialize a random process $\mathcal{R}$ for exploration;
6: Receive a initial observed state $\mathbf{s}_1$;
   /**Decision Epoch**/
7: for t = 1 to $T$ do
8:     Derive proto-action $\hat{\mathbf{a}}$ from the actor network $f(\cdot)$;
9:     Apply exploration policy to $\hat{\mathbf{a}}$: $\mathcal{R}(\hat{\mathbf{a}}) = \hat{\mathbf{a}} + \epsilon\mathbf{I}$;
10:    Find K-NN actions $\mathbf{A}_K$ of $\hat{\mathbf{a}}$ by solving a series of MIQP-NN problems (described above);
11:    Select action $\mathbf{a}_t = \mathrm{argmax}_{\mathbf{a}\in\mathbf{A}_K} Q(\mathbf{s}_t, \mathbf{a})$;
12:    Execute action $\mathbf{a}_t$ by deploying the corresponding scheduling solution, and observe the reward $r_t$;
13:    Store transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ into **B**;
14:    Sample a random mini-batch of $H$ transition samples $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ from **B**;
15:    $y_i := r_i + \gamma \max_{\mathbf{a}\in\mathbf{A}_{i+1,K}} Q'(\mathbf{s}_{i+1}, \mathbf{a})$, $\forall i \in \{1, \cdots, H\}$, where $A_{i+1,K}$ is the set of K-NN of $f'(\mathbf{s}_{i+1})$;
16:    Update the critic network $Q(\cdot)$ by minimizing the loss:
       $L(\boldsymbol{\theta^Q}) = \frac{1}{H}\sum_{i=1}^{H}[y_i - Q(\mathbf{s}_i, \mathbf{a}_i)]^2$;
17:    Update the weights $\boldsymbol{\theta^\pi}$ of actor network $f(\cdot)$ using the sampled gradient:
       $\nabla_{\boldsymbol{\theta^\pi}} f \approx \frac{1}{H}\sum_{i=1}^{H} \nabla_{\hat{\mathbf{a}}} Q(\mathbf{s}, \hat{\mathbf{a}})|_{\hat{\mathbf{a}}=f(\mathbf{s}_i)} \cdot \nabla_{\boldsymbol{\theta^\pi}} f(\mathbf{s})|_{\mathbf{s}_i}$;
18:    Update the corresponding target networks:
       $\boldsymbol{\theta^{Q'}} := \tau\boldsymbol{\theta^Q} + (1-\tau)\boldsymbol{\theta^{Q'}}$;
       $\boldsymbol{\theta^{\pi'}} := \tau\boldsymbol{\theta^\pi} + (1-\tau)\boldsymbol{\theta^{\pi'}}$
19: end for
```

We formally present the actor-critic-based method for scheduling as Algorithm 2. First, the algorithm randomly initializes all the weights $\boldsymbol{\theta^\pi}$ of actor network $f(\cdot)$ and $\boldsymbol{\theta^Q}$ of critic network $Q(\cdot)$ (line 1). If we directly use the actor and critic networks to generate the training target values $< y_i >$ (line 15), it may suffer from unstable and divergence problems as shown in paper [64]. Thus, similar as in [64, 59], we create the target networks to improve training stability. The target networks are clones of the original actor or critic networks, but the weights of the target networks $\boldsymbol{\theta}^{Q'}$ and $\boldsymbol{\theta}^{\pi'}$ are slowly updated, which is controlled by a parameter τ. In our implementation, we set $\tau = 0.01$.

To robustly train the the actor and critic networks, we adopt the experience replay buffer **B** [41]. Instead of training network using the transition sample immediately collected at each decision epoch t (from line 8 to line 12), we first store the sample into a replay buffer **B**, then randomly select a mini-batch of transition samples from **B** to train the actor and critic networks. Note that since the size of **B** is limited, the oldest sample will be discarded when **B** is full. The sizes of replay buffer and mini-batch were set to $|\mathbf{B}| = 1000$ and $H = 32$ respectively in our implementation.

The online exploration policy (line 9) is constructed as $\mathcal{R}(\hat{\mathbf{a}}) = \hat{\mathbf{a}} + \epsilon\mathbf{I}$, where ϵ is an adjustable parameter just as the ϵ in the ϵ-greedy method [44], which determines the probability to add a random noise to the proto-action rather than take the derived action from the actor network. ϵ decreases with decision epoch t, which means with more training, more derived actions (rather than random ones) will be taken. In this way, ϵ can tradeoff exploration and exploitation. The parameter **I** is a uniformly distributed random noise, each element of which was set to a random number in $[0, 1]$ in our implementation.

The critic network $Q(\cdot)$ is trained by the mini-batch samples from **B** as mentioned above. For every transition sample $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ in the mini-batch, first we obtain the proto-action $\hat{\mathbf{a}}_{i+1}$ of the next state $\mathbf{s}_{i+1}$ from the target actor network $f'(\mathbf{s}_{i+1})$; second, we find K-NN actions $\mathbf{A}_{i+1,K}$ of the proto-action $\hat{\mathbf{a}}_{i+1}$ by solving a series of MIQP-NN problems presented above; then we obtain the highest Q-value from the target critic network, $\max_{\mathbf{a}\in\mathbf{A}_{i+1,K}} Q'(\mathbf{s}_{i+1}, \mathbf{a})$. To train critic network $Q(\mathbf{s}_i, \mathbf{a}_i)$, the target value y_i for input $\mathbf{s}_i$ and $\mathbf{a}_i$ is given by the sum of the immediate reward r_i and the discounted max Q-value (line 15). The discount factor $\gamma = 0.99$ in our implementation. A common loss function $L(\cdot)$ is used to train the critic network (line 16). The actor network $f(\cdot)$ is trained by the deterministic policy gradient method [54] (line 17). The gradient is calculated by the chain rule to obtain the expected return from the transition samples in the mini-batch with respect to the weights $\boldsymbol{\theta}^{\pi}$ of the actor network.

The actor and critic networks can be pre-trained by the historical transition samples, so usually the offline training (line 4) is performed first, which is almost the same as online learning (lines 13–18). In our implementation, we first collected $10,000$ transition samples with random actions for each experimental setup and then pre-trained the actor and critic networks offline. In this way, we can explore more possible states and actions and significantly speed up online learning.

3.2.3 Implementation Details

We implemented the proposed framework based on Apache Storm [3]. In our implementation, the custom scheduler runs within Nimbus, which has access to various information regarding executors, supervisors and slots. A socket is implemented for communications between the custom scheduler and the DRL agent. When an action is generated by the DRL agent, it is translated to a Storm-recognizable scheduling solution and pushed to the custom scheduler. Upon receiving a scheduling solution, the custom scheduler first frees the executors that need to be re-assigned and then adds them to the slots of target machines. Note that during the deployment of a new scheduling solution, we try to make a minimal impact to the DSDPS by only re-assigning those executors whose assignments are different from before while keeping the rest untouched (instead of deploying the new scheduling solution from scratch by freeing all the executors first and assigning executors one by one as Storm normally does). In this way, we can reduce overhead and make the system to re-stabilize quickly. In addition, to make sure of accurate data collection, after a scheduling solution is applied, the proposed framework waits for a few minutes until the system re-stabilizes to collect the average tuple processing time and takes the average of 5 consecutive measurements with a 10-second interval.

3.3 Performance Evaluation

In this section, we describe experimental setup, followed by experimental results and analysis.

3.3.1 Experimental Setup

We implemented the proposed DRL-based control framework over Apache Storm [3] (obtained from Storm's repository on Apache Software Foundation) and installed the system on top of Ubuntu Linux 12.04. We also used Google's TensorFlow [69] to implement and train the DNNs. For performance evaluation, we conducted experiments on a cluster in our data center. The Storm cluster consists of 11 IBM blade servers (1 for Nimbus and 10 for worker machines) connected by a 1Gbps network, each with an Intel Xeon Quad-Core 2.0GHz CPU and 4GB memory. Each worker machine was configured to have 10 slots.

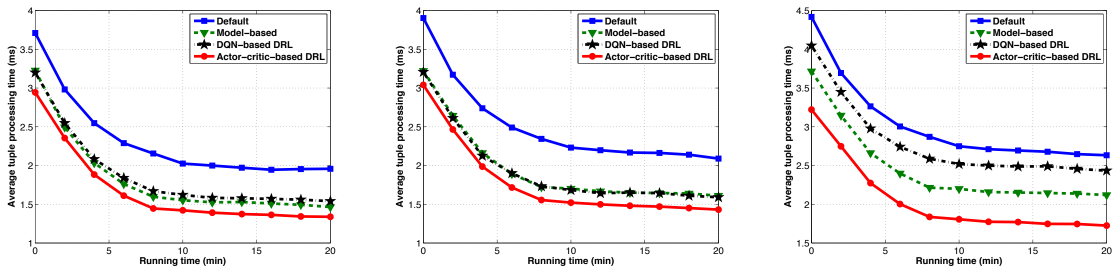
We implemented three representative DSDPS applications (called topologies in Storm) to test the proposed framework: continuous queries, log stream processing and word count (stream version), which are described in the following. We chose these three applications because they are considered the most popular and representative SDP applications. Specifically, continuous queries is one of the most popular SDP applications on Storm; log stream processing is another popular SDP application that has been introduced in the well-known Storm guidebook [65]; and word count is the most widely-used benchmark application for big data processing. Moreover, these applications were also used for performance evaluation in [66] that presented the state-of-the-art model-based approach; thus using them ensures fair comparisons.

Continuous Queries Topology (Figure 2.6): This topology represents a popular application on Storm (described in Section 2.3). To perform a comprehensive evaluation, we came up with 3 different setups for this topology: small-scale, medium-scale and large-scale. In the small-scale experiment, a total of 20 executors were created, including 2 spout executors, 9 Query bolt executors and 9 File bolt executors. In the medium-scale experiment, we had 50 executors in total, including 5 spout executors, 25 Query bolt executors and 20 File bolt executors. For the large-scale experiment, we had a total of 100 executors, including 10 spout executors, 45 Query bolt executors and 45 File bolt executors.

Log Stream Processing Topology (Figure 2.5): The topology is detailedly described in Section 2.3. In our experiment, the topology was configured to have a total of 100 executors, including 10 spout executors, 20 LogRules bolt executors, 20 Indexer bolt executors, 20 Counter bolt executors and 15 executors for each Database bolt.

Word Count Topology (stream version) (Figure 2.4): This topology is described in Section 2.3. In the experiment, the topology was configured to have a total of 100 executors, including 10 spout executors, 30 SplitSentence bolt executors, 30 WordCount executors and 30 Database bolt executors.

3.3.2 Experimental Results and Analysis



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

Fig. 3.3: Average tuple processing time over the continuous queries topology

In this section, we present and analyze experimental results. To well justify effectiveness of our design, we compared the proposed DRL-based control framework with the actor-critic-based method (labeled as “Actor-critic-based DRL") with the default scheduler of Storm (labeled as “Default") and the state-of-the-art model-based method proposed in a very recent paper [66] (labeled as “Model-based") in terms of average (end-to-end) tu-

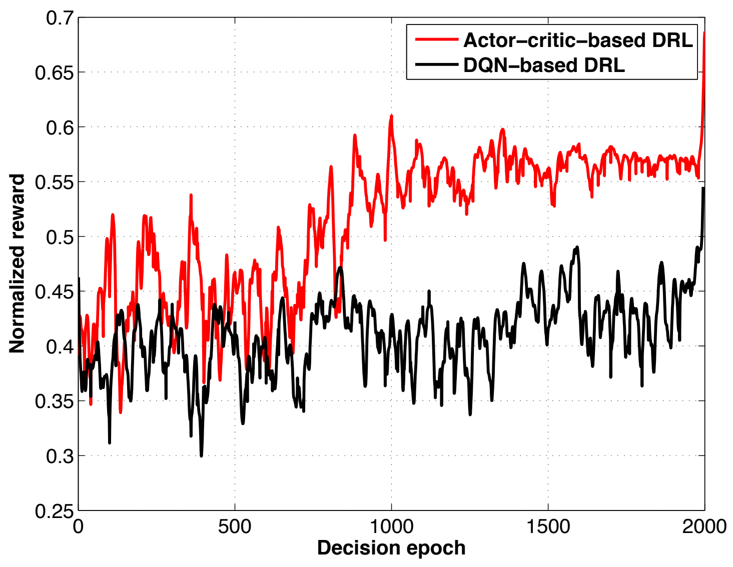


Fig. 3.4: Normalized reward over the continuous queries topology (large-scale)

ple processing time. Moreover, we included the straightforward DQN-based DRL method (described in Section 3.2.2) in the comparisons (labeled as “DQN-based DRL").

For the proposed actor-critic-based DRL method and the DQN-based DRL method, both the offline training and online learning were performed to train the DNNs to reach certain scheduling solutions, which were then deployed to the Storm cluster described above. The figures presented in the following show the average tuple processing time corresponding to the scheduling solutions given by all these methods in the period of 20 minutes. In addition, we show the performance of the two DRL methods over the online learning procedure in terms of the reward. For illustration and comparison purposes, we normalize and smooth the reward values using a commonly-used method $\frac{r-r_{\min}}{r_{\max}-r_{\min}}$ (where r is the actual reward, $r_{\min}$ and $r_{\max}$ are the minimum and maximum rewards during online learning respectively) and the well-known forward-backward filtering algorithm [70] respectively. Note that in Figures 3.3, 3.5 and 3.7, time 0 is the time when a scheduling solution given by a *well-trained* DRL agent is deployed in the Storm. It usually takes a little while (10-20 minutes) for the system to gradually stabilize after a new scheduling solution is deployed. This process is quite smooth, which has also been shown in [66]. So these figures do not show the performance of the DRL methods during training processes but the performance of the scheduling solutions given by well-trained DRL agents. The rewards given by the DRL agents during their online learning processes are shown in Figures 3.4, 3.6 and 3.8. This process usually involves large fluctuations, which have also been shown in other DRL-related works such as [61, 59].

Continuous Queries Topology (Figure 2.6): We present the corresponding experimental results in Figures 3.3 and 3.4. As mentioned above, we performed experiments on this topology using three setups: small-scale, medium-scale and large-scale, whose corresponding settings are described in the last subsection.

From Figure 3.3, we can see that for all 3 setups and all the four methods, after a scheduling solution is deployed, the average tuple processing time decreases and stabilizes at a lower value (compared to the initial one) after a short period of $8-10$ minutes. Specifically, in Figure 3.3(a) (small-scale), if the default scheduler is used, it starts at 3.71ms and stabilizes at 1.96ms; if the model-based method is employed, it starts at 3.22ms and stabilizes at 1.46ms; if the DQN-based DRL method is used, it starts at 3.20ms and stabilizes at 1.54ms; and if the actor-critic-based DRL method is applied, it starts at 2.94ms and stabilizes at 1.33ms. In this case, the actor-critic-based DRL method reduces the average tuple processing time by 31.4% compared to the default scheduler and by 9.5% compared to the model-based method. The DQN-based DRL method performs slightly worse than the model-based method.

From Figure 3.3(b) (medium-scale), we can see that the average tuple processing times given by all the methods slightly go up. Specifically, if the default scheduler is used, it stabilizes at 2.08ms; if the model-based method is employed, it stabilizes at 1.61ms; if the DQN-based DRL method is used, it stabilizes at 1.59ms; and if the actor-critic-based DRL method is applied, it stabilizes at 1.43ms. Hence, in this case, the actor-critic-based DRL method achieves a performance improvement of 31.2% over the default scheduler and 11.2% over the model-based method. The performance of DQN-based DRL method is still comparable to the model-based method.

From Figure 3.3(c) (large-scale), we can observe that the average tuple processing times given by all the methods increase further but still stabilize at reasonable values, which essentially shows that the Storm cluster undertakes heavier workload but has not been overloaded in this large-scale case. Specially, if the default scheduler is used, it stabilizes at 2.64ms; if the model-based method is employed, it stabilizes at 2.12ms; if the DQN-based DRL method is used, it stabilizes at 2.45ms; and if the actor-critic-based DRL method is applied, it stabilizes at 1.72ms. In this case, the actor-critic-based DRL method achieves a more significant performance improvement of 34.8% over the default scheduler and 18.9% over the model-based method. In addition, the performance of the DQN-based DRL method is noticeably worse than that of the model-based method.

In summary, we can make the following observations: 1) The proposed actor-critic-based DRL method consistently outperforms all the other three methods, which well justifies the effectiveness of the proposed model-free approach for control problems in DSDPSs. 2) The performance improvement (over the default schedule and the model-based method) offered by the proposed actor-critic-based DRL method become more and more significant with the increase of input size, which shows that the proposed model-free method works even better when the distributed computing environment becomes more and more sophisticated. 3) Direct application of DQN-based DRL method does not work well, especially in the large case. This method lacks a carefully-designed mechanism (such as the proposed MIQP-based mechanism presented in Section 3.2.2) that can fully discover the action space and make a wise action selection. Hence, in large cases with huge action spaces, random selection of action may lead to a suboptimal or even poor decision.

We further exploit how the two DRL methods behave during online learning by showing how the normalized reward varies over time within $T = 1500$ decision epochs in Figure 3.4. We performed this experiment using the large-scale setup described above. From this figure, we can observe that both methods start from similar initial reward values. The DQN-based DRL method keeps fluctuating during the entire procedure and ends at an average award value of 0.44 (the average over the last 200 epochs); while the actor-critic-based DRL method experiences some fluctuations initially, then gradually climbs to a higher value. More importantly, the actor-critic-based DRL method consistently offers higher rewards compared to the DQN-based method during online learning. These results further confirm superiority of the proposed method during online learning. Moreover, we find that even in this large-scale case, the proposed actor-critic-based DRL method can quickly (within only 1500 decision epochs) reach a good scheduling solution (whose performance have been discussed above) without going though a really long online learning procedure (e.g., several million epochs) shown in other applications [41]. This justifies the practicability of the proposed DRL method for online control in DSPDSs.

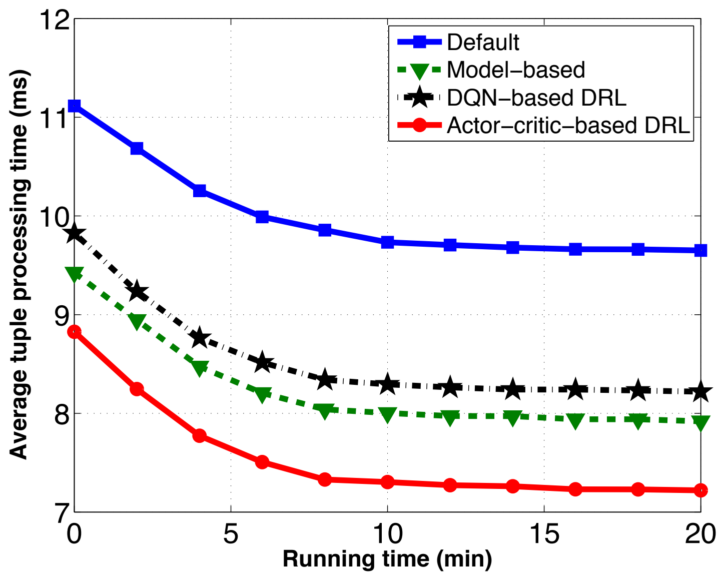


Fig. 3.5: Average tuple processing time over the log processing topology (large-scale)

Log Stream Processing Topology (Figure 2.5): We performed a large-scale experiment over the log stream processing topology, whose settings have been discussed in the last subsection too. We show the corresponding results in Figures 3.5 and 3.6. This topology is more complicated than the previous continuous queries topology, which leads to a longer average tuple processing time no matter which method is used.

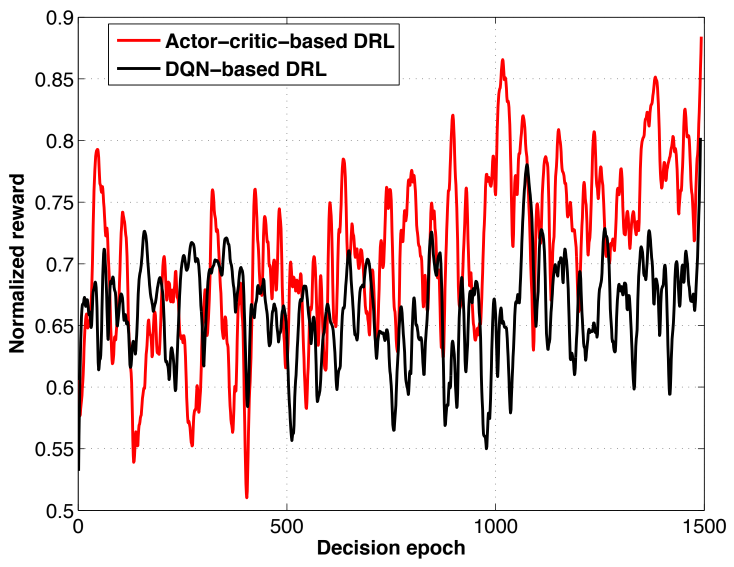


Fig. 3.6: Normalized reward over the log processing topology (large-scale)

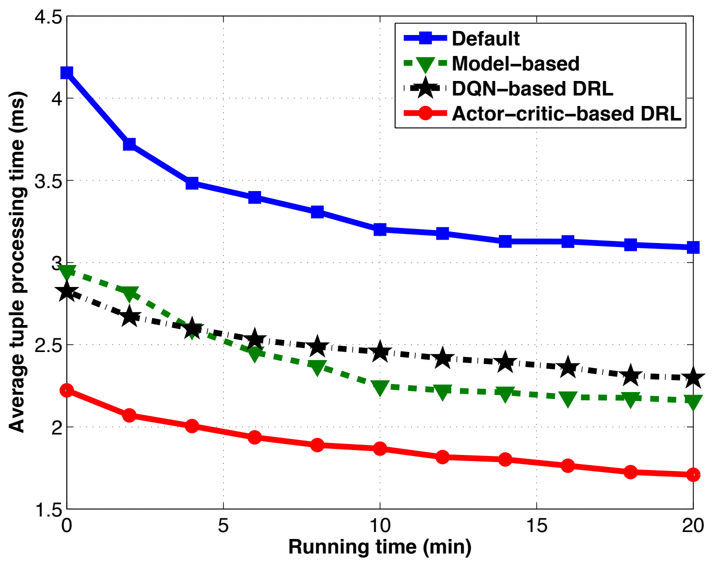


Fig. 3.7: Average tuple processing time over the word count topology (large-scale)

In Figure 3.5, if the default scheduler is used, it stabilizes at 9.61ms; if the model-based method is employed, it stabilizes at 7.91ms; if the DQN-based DRL method is used, it stabilizes at 8.19ms; and if the actor-critic-based DRL method is applied, it stabilizes at 7.20ms. As expected, the proposed actor-critic-based DRL method consistently outperforms the other three methods. Specifically, it reduces the average tuple processing time by

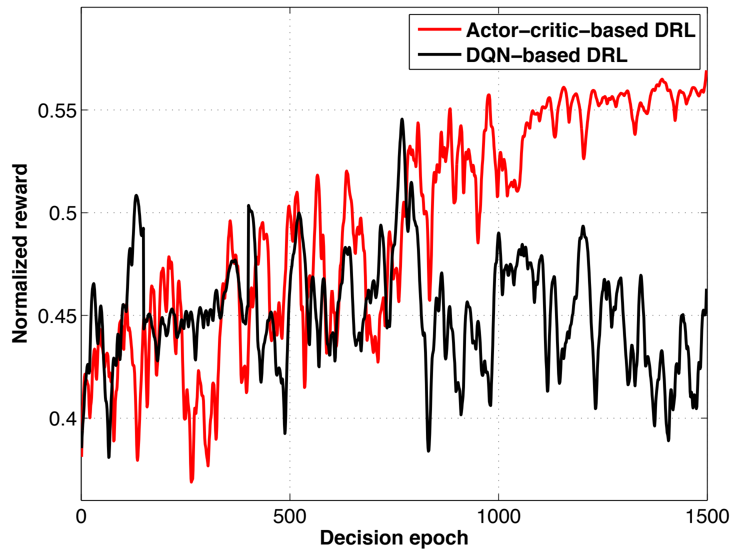


Fig. 3.8: Normalized reward over the word count topology (large-scale)

25.1% compared to the default scheduler and 9.0% compared to the model-based method. Furthermore, the DQN-based DRL method performs worse than the model-based method, which is consistent with the results related to the continuous queries topology. Similarly, we show how the normalized reward varies over time within $T = 1500$ decision epochs in Figure 3.6. From this figure, we can make a similar observation that the actor-critic-based DRL method consistently leads to higher rewards compared to the DQN-based method during online learning. Obviously, after a short period of online learning, the proposed actor-critic-based DRL method can reach a good scheduling solution (discussed above) in this topology too.

Word Count Topology (stream version) (Figure 2.4): We performed a large-scale experiment over the word count (stream version) topology and show the corresponding results in Figures 3.7 and 3.8. Since the complexity of this topology is similar to that of the continuous queries topology, all the four methods give similar average time processing times.

In Figure 3.7, if the default scheduler is used, it stabilizes at 3.10ms; if the model-based method is employed, it stabilizes at 2.16 ms; if the DQN-based DRL method is used, it stabilizes at 2.29ms; and if the actor-critic-based DRL method is applied, it stabilizes at 1.70ms. The actor-critic-based DRL method results in a performance improvement of 45.2% over the default scheduler and 21.3% improvement over the model-based method. The performance of the DQN-based DRL method is still noticeably worse than the model-based method. We also show the performance of the two DRL methods during online learning in Figure 3.8, from which we can make observations similar to those related to the first two topologies.

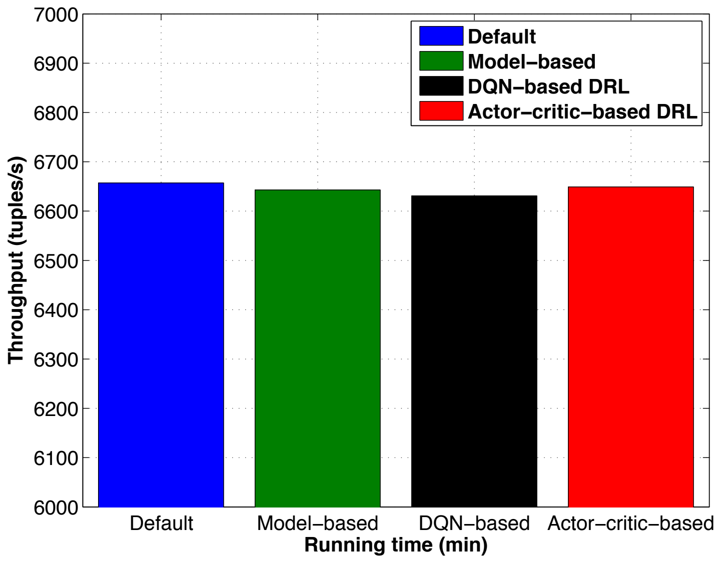
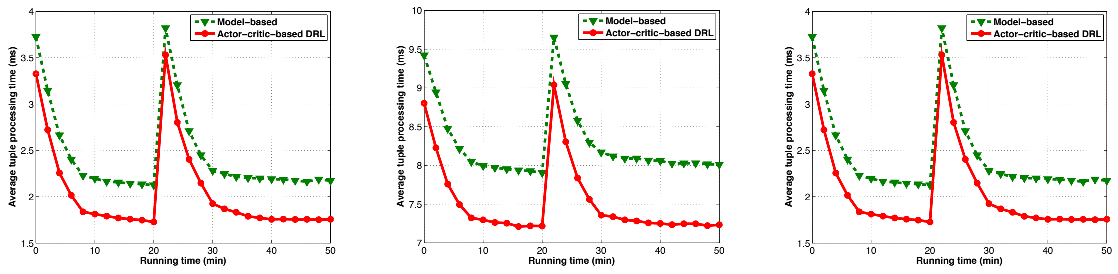


Fig. 3.9: Average throughput over the continuous queries topology (large-scale)



(a) Continuous queries topology (b) Log stream processing topology (c) Word count topology

Fig. 3.10: Average tuple processing time over 3 different topologies (large-scale) under significant workload changes

In addition, we show the average throughput given by the four methods over the continuous queries topology (large-scale) in Figure 3.9. From this figure, we can see that all the methods show similar throughput (about 6600-6700 tuples per second), which are fairly close to the data arrival rate at the spout.

Moreover, we compare the proposed model-free method with the model-based method under significant workload changes on 3 different topologies (large-scale). For the continuous queries topology (large-scale), we can see from Figure 3.10(a) that when the workload is increased by 50% at 20 minute, the average tuple processing time of the actor-critic-based DRL method rises sharply to a relatively high value then gradually stabilizes at 1.76ms, while the model-based method rises sharply too and then stabilizes at 2.17ms. The spikes are caused by the adjustment of the scheduling solution. However, we can observe that once the system stabilizes, the proposed method leads to a very minor increase on average tuple processing time. Hence, it is sensitive to the workload change and can quickly adjust its scheduling solution accordingly to avoid performance degradation. This result well justifies the robustness of the proposed method in a highly dynamic environment with significant workload changes. Moreover, during the whole period, we can see that the proposed method still consistently outperforms the model-based method. We can make similar observations for the other two topologies from Figures 3.10(b)–3.10(c).

3.4 Summary

In this chapter, we investigated a novel model-free approach that can learn to well control a DSDPS from its experience rather than accurate and mathematically solvable system models, just as a human learns a skill. We presented design, implementation and evaluation of a novel and highly effective DRL-based control framework for DSDPSs, which minimizes the average end-to-end tuple processing time. The proposed framework enables model-free control by jointly learning the system environment with very limited runtime statistics data and making decisions under the guidance of two DNNs, an actor network and a critic network. We implemented it based on Apache Storm, and tested it with three representative applications: continuous queries, log stream processing and word count (stream version). Extensive experimental results well justified the effectiveness of our design, which showed: 1) The proposed framework achieves a performance improvement of 33.5% over Storm's default scheduler and 14.0% over the state-of-the-art model-based method on average. 2) The proposed DRL-based framework can quickly reach a good scheduling solution during online learning.

Chapter 4 Model-free Scheduling for DSDPSs with a Variable Number of Threads

4.1 Overview

In this chapter, we present design, implementation and evaluation of a control framework, EXTRA (EXperience-driven conTRol frAmework), for scheduling in DSDPSs. Our design is novel and unique due to the following reasons. In summary, we made the following contributions:

- 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 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.

- 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.

4.2 Design and Implementation of the Proposed Framework

In this section, we present the design and implementation details of the proposed framework.

4.2.1 Overview of Our Design

The proposed framework, EXTRA, consists of the following 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 4.2.2) 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 4.2.2).

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.

Our design leads to several nice 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 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 4.2.2), which gradually discovers the best way to control a DSPDS from its own experience; while the current practice employs a rather trivial method that evenly distributes workload over machines. In addition, we strongly emphasize user transparency (Section 4.2.3) 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.

4.2.2 DRL-based Scheduling with a Variable Number of Threads

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 4.1: 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 $\mathcal{M}$, a set of processes $\mathcal{P}$, and a set of threads $\mathcal{N}$, a scheduling solution specifies how to assign each thread to a process of a machine, i.e., two mappings: $\mathcal{N} \mapsto \mathcal{P}$ and $\mathcal{P} \mapsto \mathcal{M}$. Similar as in [9, 66], 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 [9]. Based on this design, the above two mappings can be merged into just one mapping: $\mathcal{N} \mapsto \mathcal{M}$, i.e., to assign each thread to a machine. 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 $\mathcal{M}$. Different scheduling solutions lead to different tuple processing and transfer delays at/between threads at runtime thus different end-to-end tuple processing times [66]. The objective of this problem is to minimize the average end-to-end tuple processing time. We have described how DRL, particularly DDPG, works in Section 1.2.3. Here we introduce how to leverage DDPG 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 $\mathcal{A}$ is the space that 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. We include workload in the state because we aim to design a DRL method that can be adaptive and sensitive to the incoming workload, which has been validated by our experimental results.

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.

The basic DRL technique, the DQN-based method [41], 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.

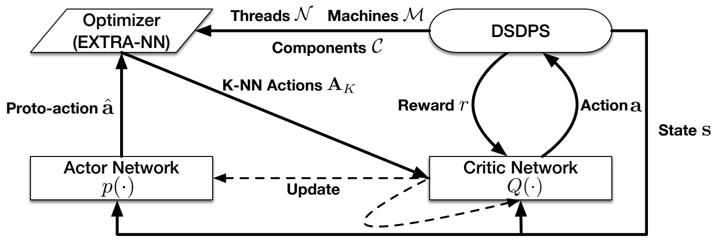


Fig. 4.1: The actor-critic-based method

In our design, we choose to leverage some advanced RL techniques, including actor-critic method [45, 64] and the deterministic policy gradient [54], for solving the scheduling problem. Note that since these techniques only provide a general framework, we still need to come up with a specific solution to our problem studied here. The basic idea of the proposed method is illustrated in Figure 4.1, which includes three major components: 1) an actor network that takes the state as input and returns a continuous *proto-action* $\hat{\mathbf{a}}$ (base solution); 2) an optimizer that finds a set $\mathbf{A}_K$ of K Nearest Neighbors (K-NN) of $\hat{\mathbf{a}}$ in the action space; and 3) a critic network that takes the state and and $\mathbf{A}_K$ as input and returns Q value for each action $\mathbf{a} \in \mathbf{A}_K$. Then an action with the highest Q value is selected for execution.

Specifically, the actor network $p(\mathbf{s}|\boldsymbol{\theta})$ is a function parameterized by $\boldsymbol{\theta}$. $\hat{\mathbf{a}}$ is returned as a proto-action that takes continuous values so $\hat{\mathbf{a}} \notin \mathcal{A}$ (i.e., not a feasible action). For the actor network, we use a 2-layer fully-connected feedforward neural network, which includes 64 and 32 neurons in the first and second layers respectively and uses the hyperbolic tangent function $\tanh(\cdot)$ for activation. This activation function is chosen because it has been shown by our empirical study that it works best among all the commonly-used activation functions.

The next and most critical step is to find the K-NN of the proto-action, which has not been well discussed in related work [64]. 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. Hence, enumerating all actions in $\mathcal{A}$ and doing a linear search for the K-NN may take exponentially long time. We introduce an optimizer, which finds the K-NN by solving a series of Mixed-Integer Quadratic Programming (MIQP) problems defined in the following:

EXTRA-NN:

$$\begin{aligned} \min_{\mathbf{a}} : \quad & \|\mathbf{a} - \hat{\mathbf{a}}\|_2^2 \\ \text{s.t.:} & \sum_{j=1}^{M} a_{ij} \geq 1, \forall i \in \{1, \cdots, C\}; \\ & \sum_{j=1}^{M} a_{ij} \leq C_i, \forall i \in \{1, \cdots, C\}; \\ & a_{ij} \in \{0, \cdots, C_i\}, \forall i \in \{1, \cdots, C\}, \forall j \in \{1, \cdots, M\}. \end{aligned} \tag{4.1}$$

Solving EXTRA-NN can find the nearest neighbor **a** of the proto-action action **â**. The constraints ensure the feasibility of action **a**, i.e., $\mathbf{a} \in \mathcal{A}$ (See the definition of an action). K-NN of **â** can be obtained by iteratively solving EXTRA-NN K times. Each time, one of the K-NN of the proto-action is returned, the corresponding values $< a_{ij} >$ are fixed, then the MIQP-NN problem is updated and solved again to obtain the next nearest neighbor until all the K-NN are obtained. In practise, this simple MIQP problem can be efficiently solved using a solver as long as C, $< C_i >$ and M are not too large. In our experiments, we solved EXTRA-NN using the Gurobi Optimizer [67] 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 [68] 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 parameterized by $\boldsymbol{\phi}$, which returns Q value for each action $\mathbf{a} \in \mathbf{A}_K$ as a DQN. The action can then be selected as follows:

$$\pi_Q(\mathbf{s}) = \underset{\mathbf{a}\in\mathbf{A}_K}{\operatorname{argmax}}\, Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\phi}). \tag{4.2}$$

Similar to the actor network $p(\cdot)$, we employ a 2-layer fully-connected feedforward neural network to serve as the critic network, which includes 64 and 32 neurons in the first and second layers respectively and employs the hyperbolic tangent function $\tanh(\cdot)$ for activation. Note that the two DNNs (actor network and critic network) are jointly trained using the collected transition samples.

We formally present the DRL-based algorithm for scheduling as Algorithm 3. Both experience replay and target networks [41, 59] are adopted in our algorithm to improve learning stability and avoid divergence. For experience replay, samples are first stored into a replay buffer **B**, then a mini-batch of transition samples are randomly selected from **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 since the size of **B** is limited, the oldest sample will be discarded when **B** is full. The sizes of experience replay buffer and mini-batch were set to $|\mathbf{B}| = 1000$ and $L = 32$ respectively in our implementation. Target networks are clones of the original actor or critic networks, but their weights $\boldsymbol{\theta'}$ and $\boldsymbol{\phi'}$ are slowly updated, which is controlled by a parameter τ (line 18). In our implementation, $\tau = 0.01$.

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 8–18). In our implementation, for

Algorithm 3 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 $\mathbf{A}_K$ containing K-NN of $\hat{\mathbf{a}}$ by solving EXTRA-NN (described above);
11:    Select action $\mathbf{a}_t = \mathrm{argmax}_{\mathbf{a}\in\mathbf{A}_K} Q(\mathbf{s}_t, \mathbf{a})$;
12:    Execute action $\mathbf{a}_t$ by deploying the corresponding scheduling solution, then ob-
       serve the reward and the new state;
13:    Store transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ into $\mathbf{B}$;
14:    Sample a random mini-batch of $L$ transition samples $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ from $\mathbf{B}$;
15:    $y_i := r_i + \gamma \max_{\mathbf{a}\in\mathbf{A}_{i+1,K}} Q'(\mathbf{s}_{i+1}, \mathbf{a})$, $\forall i \in \{i, \cdots, L\}$, where $A_{i+1,K}$ is the set of
       K-NN of $p'(\mathbf{s}_{i+1})$;
       /**Updating the networks**/
16:    Update $\boldsymbol{\phi}$ to minimize the loss given by Equation (1.2);
17:    Update $\boldsymbol{\theta}$ using the sampled gradient given by Equation (1.4);
18:    Update $\boldsymbol{\theta'}$ and $\boldsymbol{\phi'}$ as follow:
       $\boldsymbol{\theta'} := \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta'}$
       $\boldsymbol{\phi'} := \tau\boldsymbol{\phi} + (1-\tau)\boldsymbol{\phi'}$;
19:  end for
20: end 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.

The online exploration policy (line 9) is constructed as $\mathcal{X}(\hat{\mathbf{a}}) = \hat{\mathbf{a}} + \epsilon\mathbf{I}$, where ϵ is an adjustable parameter just as the ϵ in the ϵ-greedy method [44], and $\mathbf{I}$ is a random noise, each element of which was set to a random number uniformly distributed in $[0, 1]$ in our implementation. 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 (rather than random ones) will be taken. A trade-off between exploration and exploitation is thus achieved.

The critic network $Q(\cdot)$ is trained by the mini-batch samples from **B** as mentioned above. For every transition sample $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ in the mini-batch, first we obtain the proto-action $\hat{\mathbf{a}}_{i+1}$ of the next state $\mathbf{s}_{i+1}$ from the target actor network $p'(\mathbf{s}_{i+1})$; second, we find K-NN actions $\mathbf{A}_{i+1,K}$ of the proto-action $\hat{\mathbf{a}}_{i+1}$ by solving EXTRA-NN presented above; then the highest Q-value from the target critic network, $\max_{\mathbf{a}\in\mathbf{A}_{i+1,K}} Q'(\mathbf{s}_{i+1}, \mathbf{a})$, is taken (line 15). To train critic network $Q(\mathbf{s}_i, \mathbf{a}_i)$, the target value y_i for input $\mathbf{s}_i$ and $\mathbf{a}_i$ is given by the sum of the immediate reward r_i and the discounted max Q-value (line 15). The discount factor $\gamma = 0.99$ in our implementation. A common loss function $L(\cdot)$ defined in Equation (1.2) is used to train the critic network (line 16). In order to obtain the expected return from the transition samples in the mini-batch regarding the weights $\boldsymbol{\theta}$ of the actor network $p(\cdot)$, chain rule is used to calculate the gradient. $p(\cdot)$ is trained using the deterministic policy gradient method given by Equation (1.4) (line 17).

4.2.3 Implementation Details

The proposed framework was implemented based on Apache Storm [3]. The DRL agent was implemented as a separate program running independently from Storm.

At each decision epoch, the DRL agent calls the Storm REST API to remotely obtain the state and reward (Section 4.2.2). Then the proposed DRL-based scheduling algorithm (Algorithm 3) 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-assign those executors whose assignments are different 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 scheduling solution is applied. After that, the average of 5 consecutive measurements is taken with a 10-second interval in between.

4.3 Performance Evaluation

In this section, we describe experimental setup, followed by experimental results and analysis.

4.3.1 Experimental Setup

We implemented EXTRA over Apache Storm [3] (obtained from Storm's repository on Apache Software Foundation) and installed the system on top of Ubuntu Linux. We also used Google's TensorFlow [69] to implement and train the DNNs. For performance evaluation, we conducted experiments on a cluster in our data center, which consists of 11 IBM blade servers (1 for Nimbus and 10 for worker nodes) connected by a 1Gbps network, each with an Intel Xeon Quad-Core 2.0GHz CPU and 4GB memory. Each worker node was configured to have 10 slots.

We implemented 3 representative SDP applications to test EXTRA: continuous queries, word count (stream version) and log stream processing, which are described in the following:

Continuous Queries Topology (Figure 2.6): This topology represents one of the most popular SDP applications (described in Section 2.3).

To perform a comprehensive evaluation, we came up with 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. In the medium-scale experiment, for each topology, we had a maximum number of 50 executors in total, including a maximum number of 5 spout executors, 25 Query bolt executors and 20 File bolt executors. For the large-scale experiment, for each topology, we had a maximum number of 100 executors in total, including a maximum number of 10 spout executors, 45 Query bolt executors and 45 File bolt executors.

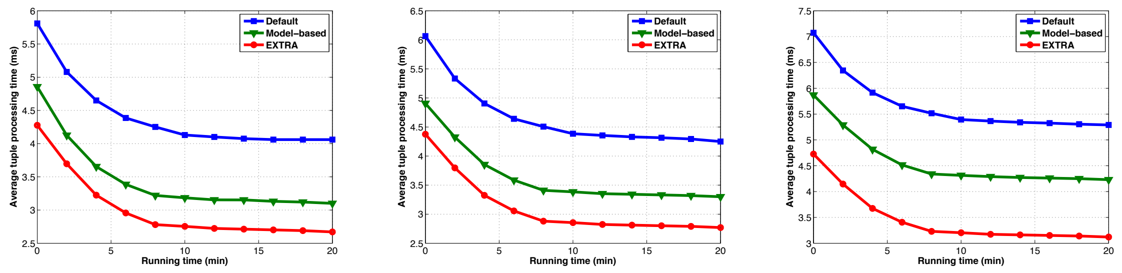
Word Count Topology (stream version) (Figure 2.4): Widely known as a classical MapReduce application, the original version of the topology counts every word's frequency of occurrence in one or multiple files. We modified it into an SDP application running a similar routine but with a stream data source (detailedly described in Section 2.3).

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.

Log Stream Processing Topology (Figure 2.5): This is another very popular SDP application (described in detail in Section 2.3).

In our 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.



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

Fig. 4.2: Average tuple processing time over 3 continuous queries topologies

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.

Varying Workload Scenario: To test how EXTRA performs in a highly dynamic environment, we also performed experiments with varying workload: 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).

4.3.2 Experimental Results and Analysis

In this section, we present and analyze experimental results. To well justify effectiveness of our design, we compared EXTRA with the state-of-the-art model-based method proposed in a very recent paper [66] (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.

For EXTRA, both offline training and online learning were performed to train the DNNs to reach certain scheduling solutions, which were then deployed to the Storm cluster described above. We present the corresponding results in Figures 4.2–4.7.

Continuous Queries Topology (Figure 2.6): 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.

From Figure 4.2, we can see that for all 3 methods, after a scheduling solution is deployed, the average tuple processing time decreases and stabilizes at a relatively low value (compared to the initial one) after a short period of $8 - 10$ minutes. Specifically, in Figure 4.2(a) (small-scale), when the default scheduler is used, it starts at 5.81ms and stabilizes at 4.06ms; when the model-based method is employed, it starts at 4.86ms and stabilizes at 3.10ms; and when EXTRA is used, it starts at 4.28ms and stabilizes at 2.67ms. In this case, EXTRA reduces the average tuple processing time by 34.2% compared to the default scheduler and by 14.0% compared to the model-based method.

From Figure 4.2(b) (medium-scale), we can see that the average tuple processing times given by all 3 methods slightly go up. Specifically, when the default scheduler is used, it stabilizes at 4.25ms; when the model-based method is employed, it stabilizes at 3.31ms; and when EXTRA is applied, it 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.

From Figure 4.2(c) (large-scale), we can observe that the average tuple processing times given by all 3 methods increase further but still stabilize at reasonable values, which essentially shows that the Storm cluster undertakes heavier workload but has not been overloaded in this large-scale case. Specifically, while the default scheduler is used, it stabilizes at 5.29ms; while the model-based method is employed, it stabilizes at 4.23ms; and while EXTRA is used, it 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) (Figure 2.4): We performed a large-scale experiment over 3 word count (stream version) topologies and show the corresponding results in Figure 4.3. Since the complexity of this topology is similar to that of the continuous queries topology, all 3 methods give similar average time processing times.

In Figure 3.7, 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.

Log Stream Processing Topology (Figure 2.5): We performed a large-scale experiment over 3 log stream processing topologies as described above. We show the corresponding results in Figure 4.4. This topology is more complicated than the continuous queries topology, which leads to a longer average tuple processing time no matter which method is used.

In Figure 4.4, 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, it reduces the average tuple processing time by 38.1% compared to the default scheduler and 23.6% compared to the model-based method.

Hybrid Scenario: For the hybrid scenario with 3 different topologies, the correspond-

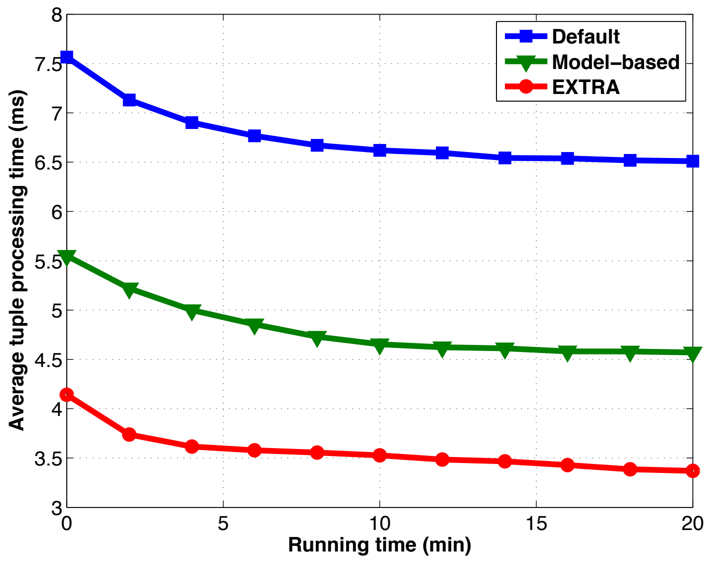


Fig. 4.3: Average tuple processing time over 3 large-scale word count topologies (stream version)

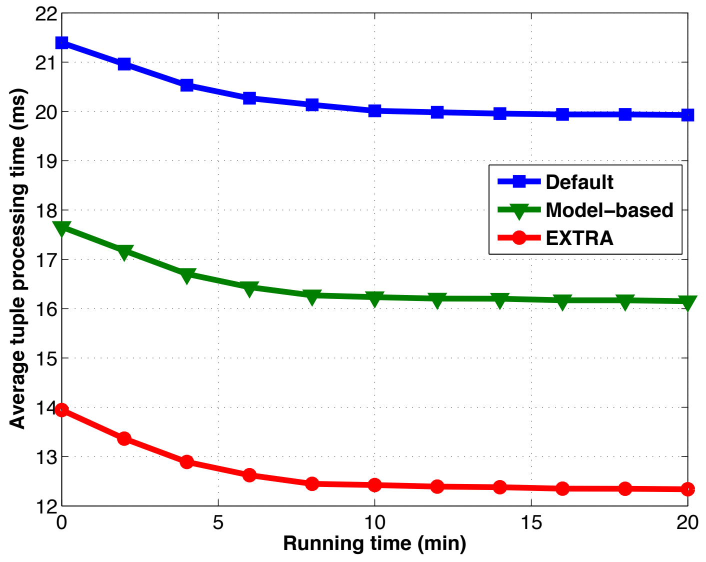


Fig. 4.4: Average tuple processing time over 3 large-scale log processing topologies

ing results are shown in Figure 4.5. 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

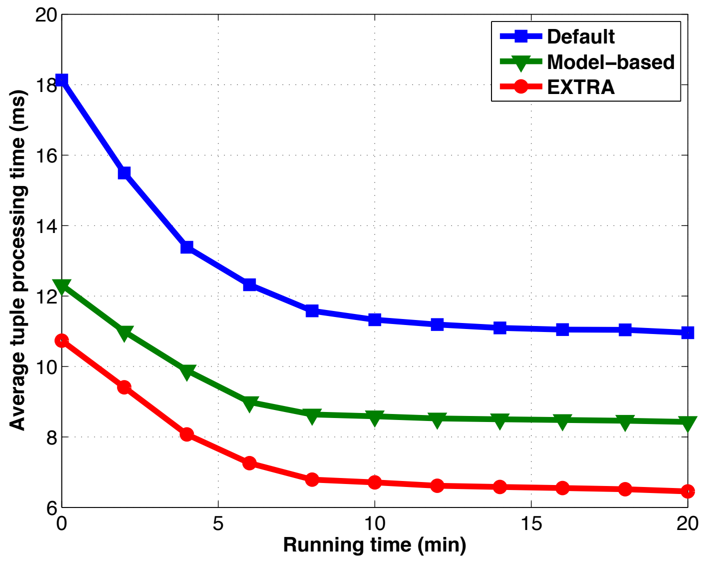


Fig. 4.5: Average tuple processing time over 3 different large-scale topologies

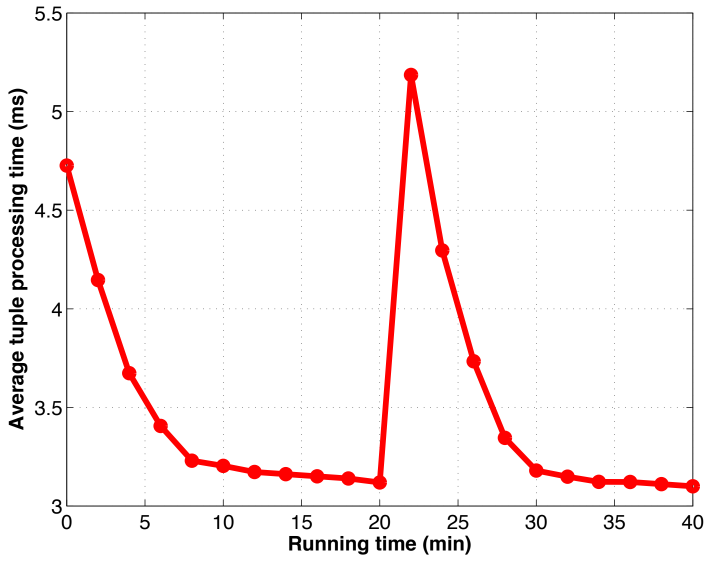


Fig. 4.6: Average tuple processing time given by EXTRA over 3 large-scale continuous queries topologies with varying workload

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

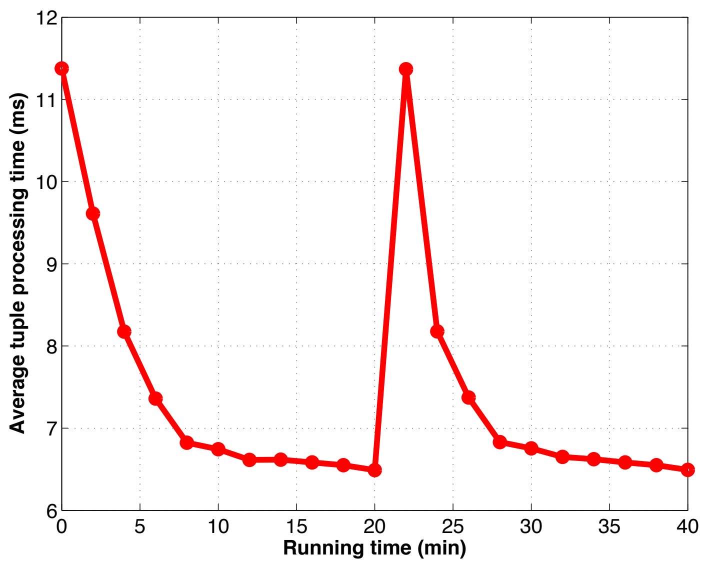


Fig. 4.7: Average tuple processing time given by EXTRA over 3 different large-scale topologies with varying workload

threads for scheduling in a DSDPS. 2) The 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 4.2.2) can lead to an efficient and effective discovery of the huge action space and thus result in a wise action selection.

Varying Workload Scenario: The corresponding results are shown in Figures 4.6 and 4.7.

In the first experiment with 3 large-scale continuous queries topologies (Fig. 4.6), during the first 20-minute period, the average tuple processing time of EXTRA stabilizes at 3.13ms. When the workload increases by 50% at 20 minute, the average tuple processing time rises sharply to a very high value then gradually stabilizes at 3.15ms respectively. The spike is caused by the adjustment of the scheduling solution. However, we can observe that once the system stabilizes, it leads to very minor increase on average tuple processing time. Hence, EXTRA is sensitive to the workload change and it can quickly adjust its schedule solution accordingly to avoid performance degradation. In the second experiment with 3 different large-scale topologies (Fig 4.7), 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 varying workload.

4.4 Summary

In this chapter, we present design, implementation and evaluation of a model-free scheduling framework with a variable number of threads, 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 a realistic setting 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 varying workload.

Chapter 5
Conclusion and Future Work

The objective of the study is to explore the possibility of discovering new approaches to achieve efficient online scheduling in Distributed Stream Data Processing Systems (DS-DPSs), which is exceptionally difficult to characterize due to the high complexity of the system environment. Currently the most widely-used solution is the round-robin manner which is fairly naive, and mathematical models such as queuing theory does not work well due to the complexity of the DSDPSs. In the dissertation, we introduced both model-based and model-free approaches to solve the problem.

For the model-based approach, we are the first to propose a method for modeling and predicting the end-to-end average tuple processing time in DSDPSs and validate the method with the widely-used open-source platform, Storm. The approach offers good prediction accuracy and significant performance improvement over Storm's default scheduler. For the model-free approach, we are the first to leverage DRL for enabling model-free control in DSDPSs. The system environment is learned with very limited runtime statistics (compared to model-based approach) and scheduling solutions are made under the guidance of DNNs. With satisfying result achieved, we believe that this simple and practical model-free approach can be easily extended to better control many other complex distributed computing systems, which has been a challenging problem in this field for years. The model-free approach with a variable number of threads enables dynamic use of a variable number of threads at runtime. This feature opens up a whole new dimension for scheduling in DSDPSs since in most of the current solutions the number of threads is decided in advance and fixed in runtime. This approach further improves flexibility as well as performance of the previous mentioned model-free approach.

It could be interesting to propose a hybrid approach, combining the model-based approach with the model-free approach together to solve the scheduling problem. The model-based approach can provide guidance for the DNNs in the model-free approach to better learn the environment. Also, although the proposed frameworks were deployed on physical machines, they can be deployed in a virtualized cloud environment by using virtual machines (instead of physical machines) to serve as worker nodes. The model-based approach will work (or can be slighted modified to work) for any DSDPS in which an application is modeled using a direct graph, and the other two model-free approaches will work for any DSDPS. Besides, we believe that the research on the DRL theory is still in its infancy. The rapid development in the DRL field will definitely offer tremendous potential in improving the performance of our proposed approaches in the future.

References

[1] J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters, *Proceedings of USENIX OSDI'2004*.

[2] Apache Hadoop, *http://hadoop.apache.org/*

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

[4] S4, *http://incubator.apache.org/s4/*

[5] 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.

[6] T. Akidau, A. Balikov, K. Bejiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom and S. Whittle, MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 2013, pp. 1033–1044.

[7] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, Timestream: reliable stream computation in the cloud, *Proceedings of EuroSys'2013*.

[8] V. Jakkula, Tutorial on support vector machine (SVM), School of EECS, Washington State University, 2006.

[9] 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.

[10] Hazelcast, *http://hazelcast.com/products/hazelcast/*

[11] PTP daemon - Precision Time Protocol daemon, *http://ptpd.sourceforge.net/*

[12] IEEE-1588 Standard, *http://www.nist.gov/el/isd/ieee/ieee1588.cfm*

[13] Storm 0.9.2 on Apache Software Foundation, *https://storm.apache.org/2014/06/25/storm092-released.html*

[14] Logstash - Open Source Log Management, *http://logstash.net/*

[15] Redis, *http://redis.io*

[16] Alice's Adventures in Wonderland, *http://www.gutenberg.org/files/11/11-pdf.pdf*

[17] 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.

[18] Apache Flink, *https://flink.apache.org/*

[19] Apache Samza, *http://samza.apache.org/*

[20] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu, Deduce: at the intersection of Mapreduce and stream processing, *Proceedings of ACM EDBT'2010*, pp. 657–662.

[21] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy and R. Sears, Online aggregation and continuous query support in MapReduce, *Proceedings of ACM SIGMOD'2010*, pp. 1115–1118.

[22] R. Karve, D. Dahiphale and A. Chhajer, Optimizing cloud MapReduce for processing stream data using pipelining, *Proceedings of European Symposium on Computer Modeling and Simulation*, 2011, pp.344–349.

[23] A. Aly, A. Sallam, B. Gnanasekaran, L. Nguyen-Dinh, W. Aref, M. Ouzzani, and A. Ghafoor, M^3: Stream Processing on Main-Memory MapReduce, *Proceedings of IEEE ICDE'2012*, pp. 1253–1256.

[24] N. Backman, K. Pattabiraman, R. Fonseca and U. Cetintemel, C-MR: continuously executing MapReduce workflows on multi-core processors, *Proceedings of MapReduce'12*.

[25] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw and N. Weizenbaum, FlumeJava: easy, efficient data-parallel pipelines, *Proceedings of ACM PLDI'2010*, pp. 363–375.

[26] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri and A. Doan, Muppet: MapReduce-style Processing of Fast Data, *Proceedings of VLDB Endowment 2012*, pp. 1814–1825.

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

[28] Spark Streaming | Apache Spark, *http://spark.apache.org/streaming/*

[29] Z. Chen, J. Xu, J. Tang, K. Kwiat and C. Kamhoua, G-Storm: GPU-enabled high-throughput online data processing in Storm, *Proceedings of IEEE BigData'2015*, pp. 307-312.

[30] Q. Jiang and S. Chakravarthy, Scheduling strategies for a data stream management system, *Technical Report CSE-2003-30*.

[31] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack and M. Stonebraker, Operator Scheduling in a Data Stream Manager, *Proceedings of VLDB'2003*.

[32] L. Aniello, R. Baldoni and L. Querzoni, Adaptive online scheduling in Storm, *Proceedings of ACM DEBS'2013*.

[33] P. Bellavista, A. Corradi, A. Reale and N. Ticca, Priority-based resource scheduling in distributed stream processing systems for big data applications, *IEEE/ACM International Conference on Utility and Cloud Computing*, 2014, pp. 363–370.

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

[35] T. Repantis and V. Kalogeraki, Hot-spot prediction and alleviation in distributed stream processing applications, *Proceedings of IEEE DSN'2008*, pp. 346–355.

[36] A. Khoshkbarforoushha, R. Ranjan, R. Gaire, P. P. Jayaraman, J. Hosking and E. Abbasnejad, Resource usage estimation of data stream processing workloads in datacenter clouds, 2015, *http://arxiv.org/abs/1501.07020*.

[37] I. Bedini, S. Sakr, B. Theeten, A. Sala and P. Cogan, Modeling performance of a parallel streaming engine: bridging theory and costs, *Proceedings of IEEE ICPE 2013*, pp. 173–184.

[38] A. Matsunaga, J. Fortes, On the use of machine learning to predict the time and resources consumed by applications, *Proceedings of IEEE CCGRID'2010*, pp. 495–504.

[39] J. Xue, F. Yan, R. Birke, L. Chen, T. Scherer and E. Smirni, PRACTISE: robust prediction of data center time series, *Proceedings of IEEE CNSM'2015*.

[40] H. Drucker, C. Burges, L. Kaufman, A. Smola and V. Vapnik, Support vector regression machines, *Proceedings of NIPS'1996*, pp. 155–161.

[41] 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. Hass-

abis, Human-level control through deep reinforcement learning, *Nature*, Vol. 518, No. 7540, 2015, pp. 529–533.

[42] 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.

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

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

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

[46] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, *Proceedings of OSDI'2008*.

[47] 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.

[48] 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.

[49] V. Konda and J. Tsitsiklis, Actor-critic algorithms, *Proceedings of NIPS'2000*, pp. 1008–1014.

[50] 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.

[51] T. Schaul, J. Quan, I. Antonoglou and D. Silver, Prioritized experience replay, *arXiv: 1511.05952*, 2015.

[52] S. Gu, T. Lillicrap, Z. Ghahramani, R. Turner and S. Levine, Q-prop: Sample-efficient policy gradient with an off-policy critic, *arXiv: 1611.02247*, 2016.

[53] V. Mnih, A. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, *Proceedings of ICML'2016*, pp. 1928–1937.

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

[55] Twitter Heron, *http://twitter.github.io/heron/*

[56] Azure Stream Analytics, *http://azure.microsoft.com/Data/StreamAnalytics/*

[57] Amazon Kinesis, *http://aws.amazon.com/kinesis/*

[58] K. Narasimhan, T. D Kulkarni and R. Barzilay, Language understanding for text-based games using deep reinforcement learning, *Proceedings of Conference on Empirical Methods in Natural Language Processing*, 2015.

[59] T. Lillicrap, 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*.

[60] J. Foerster, Y. Assael, N. Freitas and S. Whiteson, Learning to communicate with deep multi-agent reinforcement learning, *Proceedings of NIPS'2016*.

[61] H. Hasselt, A. Guez, and D. Silver, Deep reinforcement learning with double Q-learning, *Proceedings of AAAI'2016*.

[62] Y. Duan, X. Chen, R. Houthooft, J. Schulman and P. Abbeel, Benchmarking deep reinforcement learning for continuous control, *Proceedings of ICML'2016*.

[63] S. Gu, T. Lillicrap, I. Sutskever and S. Levine, Continuous deep Q-Learning with model-based acceleration, *Proceedings of ICML'2016*.

[64] G. Arnold, R. Evans, H. 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.

[65] Q. Anderson, Storm real-time processing cookbook, *PACKT Publishing*, 2013.

[66] 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.

[67] Gurobi Optimizer, *http://www.gurobi.com/*

[68] S. Boyd and L. Vandenberghe, Convex Optimization *Cambridge University Press*, 2004.

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

[70] F. Gustafsson, Determining the initial states in forward-backward filtering, *IEEE Transactions on Signal Processing*, Vol. 44, No. 4, 1996, pp. 988–992.

[71] 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.

[72] 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.

[73] 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*.

Teng Li

Syracuse University
Department of EECS
118 Remington Ave.
Syracuse, New York 13210

tli01@syr.edu
+1 (315) 806-5366

Education

Syracuse University, Syracuse, NY	**Aug 2013 - May 2018**
Ph.D. in Electrical & Computer Engineering	
Syracuse University, Syracuse, NY	**Aug 2011 - May 2013**
M.S. in Computer Engineering	
Beijing Univ. of Posts & Tele., Beijing, China	**Sep 2010 - Jun 2011**
M.S. student in Computer Science	
Beijing Univ. of Posts & Tele., Beijing, China	**Sep 2006 - Jun 2010**
B.S. in Applied Mathematics	

Interests

Stream Processing, Parallel Computing, Distributed Computing

Publications

[1] 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.
[2] T. Li, J. Tang, and J. Xu, A predictive scheduling framework for fast and distributed stream data processing, IEEE BigData'2015, pp. 333-338.
[3] T. Li, Z. Xu, J. Tang and Y. Wang, Model-free control for distributed stream data processing using deep reinforcement learning, accepted by VLDB'2018, http://arxiv.org/abs/1803.01016.
[4] X. Sheng , J. Tang, J. Wang, T. Li, G. Xue and D. Yang, LIPS: lifestyle learning via mobile phone sensing, IEEE Globecom'2016.