University of Macedonia
Department of Applied Informatics

Doctorate in Computer Science

Parallel and distributed processing of big data streams and scheduling algorithms

Nicoleta Tantalaki

Supervisor: Stavros Souravlas, Assistant Professor, University of Macedonia
Advisors: Manos Roumeliotis, Professor, University of Macedonia,
Konstantinos Psanis, Associate Professor, University of Macedonia

February, 2021

Parallel and distributed processing of big data streams and scheduling algorithms

E-mail: nicoleta@uom.gr
Department of Applied Informatics, University of Macedonia

A thesis submitted for the degree of *Doctor of Philosophy*. February, 2021

Acknowledgements

There are many people without whom this thesis would not have been possible, and I would like to thank all of them. First, I would like to express my earnest appreciation to Prof. Stavros Souravlas, my mentor and supervisor, for his guidance, patience, inspiration, and friendship. Prof. Souravlas is a great advisor and I am very thankful for his constant support and valuable help during these years. I really appreciate his knowledge in computer science, brilliant mind and skill in leading me to the threshold of my mind. He never stopped helping me in improving my scientific, technical, and writing skills. All my achievements would not have been possible if not supported by him.

I am also grateful to Prof. Manos Roumeliotis for his insightful comments and encouragement. Prof. Roumeliotis, together with Prof. Souravlas, encouraged me towards the PhD studies and created a very pleasurable and genuine environment, that initiated and fostered my passion in doing research. I must also acklowledge Prof. Konstantinos Psanis, as his comments and remarks were essential for the quality improvement of this thesis.

Moreover, I am thankful to Lauren Jones, Patrycja Lis and the rest of the Google's talent outreach team that hosted my visit to conferences in the UK and the Netherlands. They opened a new world to me, and helped me get informed about modern technological advancements and current trends in computer science. My participation in their program initially triggered my will for research and then widened my research thoughts from various perspectives. I also enjoyed the possibility of meeting very interesting people and having fruitful conversations with them.

I am also grateful to all my friends with whom I shared ideas but most importantly the laughters I needed to keep up working even under demanding conditions. In particular, I would like to thank George Kalpakis, senior research associate at the Centre for Research and Technology Hellas, for sharing his expertise and knowledge, Zacharoula Papamitsiou, senior researcher at the Norwegian University of Science and Technology, for the motivation and her invaluable feedback, and Konstantinos Chalkias, cryptographer at Facebook, for giving me the necessary inspiration at the beginning of my studies.

Most importantly, I would like to thank my parents, Charalampos and Georgia, that made me who I am, love seeking knowledge and truth, for their infinite support through my entire life and studies. I would also like to express my unconditional thankfulness to my brother Themis, and to Constantinos who incessantly believed in my efforts, for their encouragement, and patience. This thesis is dedicated to these people.

Contents

Abstract

Nowadays, we are witnessing the development of the so-called Internet of Things (IoT), where devices collect data and exploit interconnectivity to transmit it for processing in the cloud. Worldwide streams are expanding continuously, resulting in an accelerating need to efficiently and timely handle these large amounts of data that arrive continuously. Cloud computing technology with superior computational power and high reliability rises as a promising solution for the challenges posed by data stream processing.

In-memory computing is used to meet performance related requirements like latency and throughput that are extremely important in Data Stream Processing (DSP) applications. Several different technologies have emerged specifically to address the challenges of processing high-volume, real-time data, exploiting on-the-fly computations. Distributed Stream Processing Systems (DSPSs) assign applications' processing tasks to the available resources and route streaming data between them. Efficient scheduling of processing tasks can reduce application latencies and eliminate network congestions. However, the available in-built scheduling techniques of DSPSs are far from optimal.

In this thesis, we need to solve the task scheduling problem which focuses on which tasks to be allocated on which resources, and controls the order of job execution. An overview of the available DSPSs is presented and a classification of the existing scheduling policies is provided. In this way, useful information about the matters to consider when designing an effective scheduling policy is revealed. Then, a general formulation of the task scheduling problem is presented and a matrix-based, linear scheme is provided. Differently from existing research efforts, that rarely consider memory utilization in their analysis, the derived scheme is performed in a memory-efficient and well-balanced manner. It takes advantage of pipelines to efficiently handle applications, where there is need for heavy communication (all-to-all) between tasks, assigned to pairs of components.

The scheme proposed in this thesis is static. However, when it comes to streams of data, the input load usually fluctuates drastically over time. Dynamic schemes use run-time adaptations and task re-scheduling to handle possible changes in the cluster but this usually results in significant downtime and performance degradation. Rather than re-configuring online the tasks' allocation, the proposed scheme handles queue waiting times efficiently and tries to maintain a stable and robust configuration by balancing load between the cluster's nodes. Of course, an adaptive version this approach would increase its performance, so this extension is left for future work.

For concreteness, this approach is illustrated based on Apache Storm semantics. The performance evaluation depicts the importance of constraining the required buffer space and achieving load balance to improve the system's performance and overcome the challenges of running DSP applications. The proposed scheme was compared to two state-of-the-art strategies; the default Storm scheduler and R-Storm. It was found to outperform both the other strategies in terms of throughput, achieving an average of 25%-45% improvement under various scenarios, mainly as a result of reduced buffering ($\approx$ 45% less memory).

At the end of this work, the contribution of real-time data processing in an application field is presented. The field of agriculture has to face difficult challenges due to numerous technological transformations used for increasing productivity and products quality. In precision agriculture, a key component is the use of IoT and various items like sensors, control systems, robotics and autonomous vehicles that produce high velocity data streams. Current advances in IoT and cloud computing have led to the development of new applications that have great potential in precision agriculture. However, several challenges arise as open research fields and future directions are revealed.

Περίληψη

Στις μέρες μας παρατηρείται μια εκρηκτική ανάπτυξη του Διαδικτύου των Πραγμάτων, όπου πλήθος φορητών συσκευών συνδέονται και χρησιμοποιούν το Διαδίκτυο, για να συλλέγουν δεδομένα και να τα μεταφέρουν σε κάποια αρχιτεκτονική νέφους, ώστε να υποστούν κατάλληλη επεξεργασία. Συνεχόμενες ροές δεδομένων σε όλο τον κόσμο αναπτύσσονται διαρκώς, δημιουργώντας επιτακτική ανάγκη να διαχειριστούμε αυτόν τον μεγάλο όγκο δεδομένων που καταφθάνει συνεχώς, έγκαιρα και αποτελεσματικά. Οι τεχνολογίες υπολογιστικής νέφους χάρη στην ισχυρή υπολογιστική δύναμη και την αξιοπιστία που παρέχουν, φαίνονται ικανές να αντιμετωπίσουν τις προκλήσεις που χαρακτηρίζουν την επεξεργασία ροών δεδομένων.

Μοντέλα υπολογιστικής μνήμης χρησιμοποιούνται, προκειμένου να επιτευχθούν απαιτήσεις απόδοσης όπως η χρονοκαθυστέρηση και η ρυθμαπόδοση, που είναι εξαιρετικά σημαντικές για κάθε εφαρμογή επεξεργασίας ροών δεδομένων (Data Steam Processing- DSP). Πληθώρα διαφορετικών τεχνολογιών έχει προκύψει, ειδικά για να αντιμετωπίσει τις προκλήσεις της επεξεργασίας υψηλού όγκου δεδομένων σε πραγματικό χρόνο, εκμεταλλευόμενη υπολογισμούς on-the-fly.Κατανεμημένα συστήματα επεξεργασίας ροών δεδομένων αναθέτουν τις επιμέρους εργασίες μιας εφαρμογής στους διαθέσιμους πόρους και δρομολογούν ροές δεδομένων μέσα από αυτές. Η αποτελεσματική δρομολόγηση των εργασιών μπορεί να μειώσει τις χρονοκαθυστερήσεις μιας εφαρμογής και να περιορίσει τη συμφόρηση στο δίκτυο. Ωστόσο, οι τεχνικές δρομολόγησης που είναι ενσωματωμένες στα διαθέσιμα DSP συστήματα δεν είναι οι βέλτιστες δυνατές.

Στην παρούσα διατριβή, γίνεται προσπάθεια επίλυσης του προβλήματος της χρονοδρομολόγησης των εργασιών σε συστήματα επεξεργασίας ροών δεδομένων. Το πρόβλημα αυτό εστιάζει στο ποιες εργασίες πρέπει να τοποθετηθούν, σε ποιους διαθέσιμους πόρους και ελέγχει τη σειρά της εκτέλεσής τους. Αρχικά, γίνεται μια επισκόπηση των διαθέσιμων συστημάτων DSP και μια κατηγοριοποίηση των διαθέσιμων τεχνικών δρομολόγησης από μελέτη σχετικής βιβλιογραφίας. Με αυτόν τον τρόπο, προέκυψαν οι παράγοντες που πρέπει να λαμβάνονται υπόψη, όταν σχεδιάζεται μια αποτελεσματική τεχνική δρομολόγησης. Έπειτα, γίνεται μοντελοποίηση του προβλήματος και παρουσιάζεται ένα γραμμικό σχήμα βασισμένο σε μετασχηματισμούς πινάκων. Σε αντίθεση με τις υπάρχουσες προτάσεις της βιβλιογραφίας που σπάνια λαμβάνουν υπόψη την κατανάλωση μνήμης στην ανάλυσή τους, το σχήμα που προτείνεται εδώ εκτελείται με έναν τρόπο που διαχειρίζεται αποτελεσματικά τη μνήμη και είναι ισορροπημένο ως προς τον φόρτο. Το σχήμα αυτό, εκμεταλλεύεται την τεχνική της διασωλήνωσης, προκειμένου να διαχειριστεί αποτελεσματικά εφαρμογές, όπου υπάρχει ανάγκη για πλήρη επικοινωνία μεταξύ των εργασιών διαφορετικών τελεστών της εφαρμογής.

Το σχήμα της παρούσας μελέτη είναι στατικό. Ωστόσο, στην περίπτωση των ροών δεδομένων, ο φόρτος εισόδου μεταβάλλεται δραστικά με την πάροδο του χρόνου. Τα δυναμικά σχήματα προσαρμόζονται και πραγματοποιούν κατάλληλες μεταβολές στη δρομολόγηση των εργασιών κατά τη διάρκεια εκτέλεσης μιας εφαρμογής, προκειμένου να διαχειριστούν αποτελεσματικά τις αλλαγές στο cluster. Κάτι τέτοιο, όμως,

οδηγεί σε σημαντικές καθυστερήσεις και μείωση της απόδοσης του συστήματος. Το προτεινόμενο σχήμα αντί να προσαρμόζει εκ νέου κατά τη διάρκεια της εκτέλεσης, την ανάθεση των επιμέρους εργασιών μιας εφαρμογής, χειρίζεται με έναν αποτελεσματικό τρόπο τις ουρές αναμονής και προσπαθεί να διατηρήσει μια σταθερή και ισχυρή ρύθμιση, ισορροπώντας τον φόρτο μεταξύ των κόμβων του cluster. Σαφώς, μια δυναμική έκδοση της παρούσας προσέγγισης θα βελτίωνε την απόδοσή της, γι'αυτό και η επέκταση αυτή, είναι μία από τις προτάσεις για μελλοντική έρευνα.

Για λόγους ευκρίνειας, η παρούσα προσέγγιση γίνεται με βάση τη σημασιολογία του συστήματος Apache Storm. Η αποτίμηση της αποδοτικότητας του σχήματος υποδεικνύει την σημασία του περιορισμού της απαιτούμενης ενδιάμεσης μνήμης και της εξισορρόπησης φόρτου στη βελτίωση της απόδοσης του συστήματος και στην αντιμετώπιση των προκλήσεων της εκτέλεσης εφαρμογών που επεξεργάζονται ροές δεδομένων. Κατά την εκτέλεση των πειραμάτων, πραγματοποιήθηκε σύγκριση του προτεινόμενου σχήματος με τον προκαθορισμένο δρομολογητή του Apache Storm, καθώς και με τον δρομολογητή R-Storm που προέκυψε από τη βιβλιογραφική ανασκόπηση. Το παρόν σχήμα ξεπέρασε σε επίπεδο ρυθμαπόδοσης και τα δύο σχήματα, παρέχοντας βελτίωση της τάξης του 25%-45% υπό διαφορετικά σενάρια, κυρίως χάρη στη μείωση χρήσης της ενδιάμεσης μνήμης (≈45% λιγότερη μνήμη).

Στο τέλος της παρούσας διατριβής, παρουσιάζεται η συνεισφορά της επεξεργασίας δεδομένων σε πραγματικό χρόνο σε ένα πεδίο εφαρμογής. Η χρήση συσκευών IoT και εργαλείων όπως τα αυτόνομα οχήματα, οι ασύρματοι αισθητήρες και οι ρομποτικές κατασκευές, που παράγουν συνεχώς ροές δεδομένων υψηλής ταχύτητας, αποτελούν κλειδί στην εφαρμογή πρακτικών γεωργίας ακριβείας. Ο πρωτογενής τομέας καλείται σήμερα να αντιμετωπίσει ιδιαίτερες προκλήσεις χάρη στην πληθώρα των τεχνολογικών μετασχηματισμών που πραγματοποιούνται, ώστε να αυξηθεί η παραγωγικότητα και η ποιότητα των παραγόμενων προϊόντων με σεβασμό στο περιβάλλον. Νέες εφαρμογές με μεγάλες δυνατότητες έχουν αρχίσει να αναπτύσσονται, προσπαθώντας να εκμεταλλευτούν την πρόοδο του Διαδικτύου των Πραγμάτων και της υπολογιστικής νέφους. Ωστόσο, οι προκλήσεις είναι πολλές και φανερώνουν νέα ανοιχτά πεδία έρευνας και μελλοντικές τάσεις.

Chapter 1

Introduction

"Time isn't the main thing. It's the only thing."-Miles Davis

Over the past 20 years data has increased in a large scale and in various fields. This tendency is accelerated by many recent developments. Devices interact with the external environment to support decision-making. The reduced cost of sensing devices and smartphones, together with the (almost) ubiquitous Internet connectivity, has fostered the wide diffusion of new pervasive services and devices. Enhanced medical devices, agricultural drones, crop yield mapping sensors, factory automation sensors, positioning and tracking sensors in road or railway transport compose an endless list of products and services. We are witnessing to the development of the so-called Internet of Things (IoT), where devices collect data and exploit interconnectivity to transmit it to be stored and processed in the cloud.

The term of big data is used exactly to refer to this increase in the volume of data that is difficult to be stored, processed and analyzed through traditional technologies. Big data is characterized by the 4 Vs, namely, volume, variety, velocity and value that clearly depict the need for technologies, which require new forms of integration to uncover fast, hidden values from large datasets that are diverse, complex and of massive scale. Industries are more than ever interested in the high potential of big data and many government agencies announce major plans to accelerate big data research and applications. As the amount of data grows, the adoption of advanced processing solutions and the need to use models to analyse it as it arrives, becomes imperative.

1.1 Big data processing

Parallel and distributed computing is a matter of predominant importance for alleviating scale and timeline challenges in big data processing. The addition of cloud technologies created new trends that achieve significant performance gains in data processing [1]. Two different processing modes are commonly used to elaborate data over distributed computing resources; the batch and the stream processing.

In **batch processing**, data is stored usually on a distributed file system and the results of processing are produced in batches. It is efficient in processing large data volumes, as I/O operations on multiple data-items are batched. Batch processing systems compute results that are derived from all the data they encompass. Jobs are set up so that they can be completed without any human interaction. However, the time spent on processing should not be an issue for the user and depending on the size of the data being processed and the computational power of the system, output can be delayed significantly [2]. MapReduce [3] is a batch-oriented data processing paradigm. Hadoop [4] is the most used open source MapReduce implementation, available by several cloud providers. It is a programming model for processing large data sets with a distributed algorithm running on many nodes that operate in parallel. In general, MapReduce is perfect for the delayed data processing but for near real-time routines it doesn't fit [2].

A broad class of data management and analysis problems requires support for processing unbounded data with low latency and high throughput. Such problems need continuous real-time processing, and sometimes require immediate action upon the arrival of incoming data streams [5]. In **stream processing** data is processed on-the-fly, i.e., without storing it, so it can produce results in a near real-time fashion. In-memory computing is used to meet the performance related requirements that are important in streaming applications [6]. The area of stream processing is not new. In 2005, Stonebraker et al. [7] defined the requirements that should be met by a real-time stream processing system to handle stream processing (or streaming) applications. Recently, the multiplication of data stream sources (sensor networks, connected devices etc) that is observed in the context of IoT, the consequent advent of the big data era and the diffusion of the cloud computing paradigm have renewed the interest in streaming applications. Many Distributed Stream Processing Systems (DSPSs) have emerged to support processing of this data when it really matters, in real-time, the time it arrives.

1.2 Research motivation

The growth of data stream sources in the context of IoT promotes the need to use models to process and analyse data as it arrives. More and more modern applications impose tighter time constraints on a particular event. The resultant analysis yields information that can provide companies with visibility into many aspects of their business and customer activity, and enables them to respond rapidly to emerging situations. Much of the data that companies receive in real time is more valuable at the time it arrives. Environmental monitoring, fraud detection, emergency response are just a few examples of applications that require timely processing of information. For instance, a public transportation company can exploit information regarding user mobility and presence of events within the city, to adjust in real time the number of buses within the city and their route. Businesses can track changes in public preference on their brands and products by continuously analyzing social media streams. There is no point in detecting a potential buyer after the user leaves the e-commerce site. There is also no point in detecting a credit card fraud after a transaction is completed [8]. As an example, EBay detects

frauds from PayPal usage by analyzing 5 million real-time transactions every day. In Linkedin's streaming infrastructure, over 2 trillion messages are processed per day by its DSPS, Samza [9]. Online machine learning is also another promising use case. There are numerous use cases making the need for stream processing imperative.

Initially, it was interesting to identify *what* can be managed and accomplished when taking advantage of this kind of data processing. Thus, an application field that relies heavily on decision making using IoT devices and real-time decision making is chosen to be examined. Smart agriculture, nowadays, has to take advantage of real-time data communication and information processing to improve production yield, mitigating the environmental effects and reducing cost. The possibilities and the perspectives of real-time big data processing in precision agriculture are examined closely to verify its applicability and usefulness, and identify future trends.

The next question that arose was *how* can this kind of data be processed and this question formed the basis of this thesis. The unpredictable characteristics and arrival patterns in streams of data pose unique challenges in processing. Challenges arise from the need to work with huge amount of data, requiring immediate stream processing capabilities. Managing in real-time incoming data that arrives continuously at volumes and high velocity, far exceeds the capabilities of individual machines. In this appealing environment, executing a streaming application demands the deployment of distributed computing resources, which will execute it in a parallel. Moreover, to meet performance related requirements like latency and throughput that are necessary in streaming applications, in-memory computing will be needed [6, 10].

Definition: The allocation of tasks into the cluster, in such a way that the completion time is minimized and the available resources are utilized in the maximum possible degree is known as task scheduling. It focuses on which tasks to be placed on which previously obtained resources, and controls the order of job execution [11, 12] and is an NP-hard problem [13, 14].

Although there is extended literature on task scheduling in batch systems like Hadoop [15, 16], the techniques used do not fit in real-time processing of streams mainly because of the difference in the computational model used in each case. In batch processing systems, computations are assigned to the nodes where the required data is stored, while in stream processing systems, most of the communicating tasks have to be placed together on one node or rack.

Several technologies that differ in the way they handle data, have emerged, specifically to address the challenges of processing high-volume, real-time data, taking advantage of the inherent characteristics of parallel and distributed computing to meet these challenges. The available in-built scheduling techniques of the DSPSs, though, are far from optimal. For instance, round robin is the strategy used as the default scheduler of Apache Storm [17] which is one of the most prominent open-source solutions for data stream processing. Round robin does not take into account the cost of moving tuples across the sequence of tasks (as defined by the user's application). Most of the heuristics

found in the literature are based on a number of different assumptions, have different optimization goals and aim at minimizing different utility functions, like latency and network usage. However, these approaches rarely consider memory consumption in their analysis and while they take into account the capability of the resources, they generally ignore the need for load balancing.

1.3 Research methodology

We are interested in deploying stream processing applications over cloud infrastructures that comprise distributed computing resources. This is a challenging task that requires:

- to understand the needs of streaming applications and the challenges of running them over distributed environments exploiting task and data parallelism;
- to identify relevant performance attributes of applications and scheduling considerations for stream processing jobs;
- to formulate efficient and effective scheduling solutions;
- to design approaches that can operate in a DSPS;

These key issues are faced, following these steps:

- collection of appropriate studies and review;
- problem identification and formulation;
- design of resolution approaches and development; and
- experimental evaluation.

In the first step, a literature search was conducted to collect, and thoroughly analyze the existing studies, to understand the characteristics of data streams, and find suitable approaches that can operate in the environment under investigation. The international databases IEEE Xplore, Science Direct, ACM, Scopus, Google Scholar, and CiteSeer were used to retrieve authoritative academic resources. These databases were chosen because of their wide coverage of relevant literature and advanced bibliometric features, such as suggesting related literature or citations. The literature between January 2010 and December 2019 was surveyed. The choice of the review period is a practical one and takes into consideration the fact that big data is a rather recent phenomenon. There was only one exception. Stonebraker's et al. [7] paper "The Requirements of Real-Time Stream Processing", goes back in 2005 but valued as a crucial prototype that still characterises modern stream processing frameworks in the big data industry.

The study was restricted to a number of papers having the highest quality and considered as the most important resources in this field. Papers with higher impact (that is, higher number of citations) were chosen. An exception is made for papers published

in 2017–2019, where papers with a small number of citations were also accepted. In this way, gray literature, though quite informative for big data industry solutions, was avoided. Nevertheless, official documentation of several existing frameworks complemented the related articles. Papers referring to matters of interest (e.g. parallel/distributed processing, stream processing, scheduling, etc.) in the cloud that do not make actual use of big data and their inherent characteristics (i.e. volume, velocity, variety) were further filtered out. Papers not referring to stream processing of big data were also filtered out, as literature is much more rich when it comes to processing of already stored big data and Hadoop MapReduce [3] scheduling enhancements. In this way, the number of papers qualified was severely restricted according to a set of constraints.

The selected literature was then analyzed in detail to extract the information relevant to this thesis' research needs. Additional literature that had not been identified so far, was retrieved in this step as well if they were referred to by 'related work.' This 'snow-ball' approach resulted in additional articles and web-items from which relevant information was extracted as well. Information was analysed and synthesised.

By exploring the latest research contributions, it was possible to:

- identify and understand the most relevant features of stream processing applications;
- identify the most representative mechanisms used by prominent DSPSs to satisfy the stream processing requirements. We focus on open-source solutions with active community support.
- identify matters to consider when designing an effective scheduling approach
- design suitable scheduling approaches that can operate in the aforementioned DSPSs.

Following the steps mentioned previously, a suitable representation of streaming applications and system resources was identified with the aim of formulating the task allocation and scheduling problem. Leveraging on linear algebra, matrix-based transformations were used to provide a solution to the problems. The derived approach reduced the required buffer size to effectively speed up processing and tried to balance load. Based on the analysis' results regarding the available DSPSs, the **operator-based model** (described in Chapter 2) and Apache Storm's [17] semantics were used to describe this work. Finally, the resulting scheduler' s efficiency was validated and its performance was compared with other two strategies from the state of the art.

1.4 Thesis contribution

The main goal of this thesis is to present, analyze, model, and develop solutions for the scheduling of streaming applications over distributed infrastructures in the cloud. A static task allocation and scheduling scheme, performed in a memory-efficient and well-balanced manner is proposed.

Although many feasible scheduling solutions can be found in the literature, they rarely consider memory consumption in their analysis. If streams are not managed carefully processing delays can become unacceptable and lead to long queues at a processing node, buffer overflows, and memory exhaustion. The approach presented in this thesis is inspired by the idea that the reduction of the required buffer space can minimize tuple losses (and possible re-submissions) and overhead delays that heavily affect system's performance. Pipelines can further help towards this direction. Moreover, load balance can also optimise the system's performance. Multiple stream processing computations should be interleaved on the same machine to reduce the number of needed connections and assure high throughput and increased performance. On the other hand, heavily used machines result in memory waste, node failures and increased network congestion [18].Elaborate orchestration over a collection of machines is needed to meet the required performance for streaming applications and reduce costs in the cloud.

The main contributions of this thesis are the following:

- Existing scheduling policies for DSPSs are classified to develop a general taxonomy that summarizes the main choices of current solutions;
- A general topology-aware, static formulation of the task allocation and scheduling problem is provided, using matrix transformations as an extension of Apache Storm. This scheme, though, is generic and could be integrated to any DSPS, and suitable for deployment and use in large-scale clusters. Moreover, while the proposed scheduler considers only static scheduling for now, it can be extended to dynamic scenarios but this is left for future work as described in Chapter 7. The resulting approach:
  - reduces the required buffer space to increase throughput and reduce the number of tuple losses;
  - reduces the inter-node communication cost to decrease the application's latency;
  - is balanced, increasing the overall system's performance;
  - is periodic, reducing the number of necessary computations and;
  - has linear complexity, determining faster computations.
- The policy presented is validated and evaluated, relying on experiments that run on system prototypes and compare its performance with the round robin strategy, that is the default Apache Storm scheduler, and the R-Storm [19] scheduler from the state of the art.

1.5 Thesis outline

The rest of this thesis is organised as follows: Chapter 2 refers to big data stream processing. The available execution models are presented, a number of issues and

requirements of streaming applications are discussed and mechanisms used to face them in the big data era are referred. Dominant frameworks for stream processing (DSPSs) are evaluated based on several functionality characteristics, and are compared to reveal reasons for selecting each candidate solution.

Chapter 3 presents the related work regarding both execution models already incorporated in prominent DSPSs, and heuristics found in literature. In this chapter, this work is positioned with respect to the state of the art, and important scheduling considerations for stream processing jobs are discussed.

In Chapter 4, the operator placement and scheduling problem is investigated. At first, the design of Apache Storm is described. Then the mathematical background of the desired scheme is presented based on Storm's semantics. The task allocation and scheduling algorithms of proposed scheme are also presented in this chapter, which also includes the proof of the scheme's linear complexity. Motivating examples are used to depict the application of the presented algorithms.

Chapter 5 reports the experiments done on this thesis. A discussion on the findings of the experiments is also included. The evaluation demonstrates the importance of constraining the required buffer space and achieving load balance to overcome the challenges of running DSP applications.

In Chapter 6, the precision agriculture paradigm is examined, as an application field of real-time big data processing. Initially, attention is paid to machine learning techniques, that mainly support decision making in precision agriculture. Their possibilities and limitations are identified. Then, the application of big data in agriculture is discussed and the most promising areas of use are revealed. Opportunities and challenges that arise along with a cost-benefit-analysis are further analyzed.

Finally, in Chapter 7, the results and contributions of this work are summarized and directions and trends for future research are indicated.

1.6 Publications

Part of the work in this thesis has previously appeared in international journals (J) and conference (C) papers.

Chapter 2: Big Data Stream Processing
J1. N. Tantalaki, S.Souravlas and M. Roumeliotis (2020)."A review on big data real-time stream processing and its scheduling techniques", *International Journal of Parallel, Emergent and Distributed Systems*, 35:5, 571-601, DOI: 10.1080/17445760.2019.1585848

Chapter 3: Related work
J2. N. Tantalaki, S. Souravlas, M. Roumeliotis and S. Katsavounis (2020)."Pipeline-Based Linear Scheduling of Big Data Streams in the Cloud", *IEEE Access*, vol. 8, 117182-117202, 2020, doi: 10.1109/ACCESS.2020.3004612
and

(J1)

Chapter 4: Background and problem formulation *and*
Chapter 5: Experimental Results
C1. N. Tantalaki, S. Souravlas, M. Roumeliotis, and S. Katsavounis (2019)."Linear Scheduling of Big Data Streams on Multiprocessor Sets in the Cloud". In *IEEE/WIC/ACM International Conference on Web Intelligence (WI '19)*. Association for Computing Machinery, New York, NY, USA, 107–115. DOI:https://doi.org/10.1145/3350546.3352507
and

(J2)

Chapter 6: Decision Making in IoT-enabled Agriculture
J3. N. Tantalaki, S. Souravlas and M. Roumeliotis (2019)."Data-Driven Decision Making in Precision Agriculture: The Rise of Big Data in Agricultural Systems", *Journal of Agricultural & Food Information*, 20:4, 344-380, DOI: 10.1080/10496505.2019.1638264
C2. N.Tantalaki, S.Souravlas and M. Roumeliotis (2017)."Big Data Tools To Support Real-Time Decision Making In Business Operations-A View to The Future of Agriculture". In *6th International Symposium & 28th National Conference on Operational Research.* 81

Chapter 2

Big Data Stream Processing

"Technology makes possibilities. Design makes solutions"-John Maeda

Data streams is not a recently developed concept but it is becoming more important in the aforementioned context of IoT. When data arrives fast and needs to be processed with real-time restrictions, processing and analyzing are tasks that traditional data-warehousing environments cannot handle easily due to high latency and cost [20]. Centralized computing systems have been around in technological computations for years. In such systems, one central computer controls the peripherals and performs complex computations. Centralized computing systems require expensive hardware to process huge volumes of data and support multiple online users concurrently. Under these circumstances, cloud computing systems, that are typically characterized by scalable and elastic resources, arose to exploit parallel and distributed processing technology. Users can share computing resources, which can be virtualized and allocated dynamically in the cloud.

Streaming applications are represented using a directed acyclic graph (DAG), where vertices are the operators and the edges are the channels for dataflow between operators [21]. The processing can go through operators in a particular order, where the operators can be chained together, but the processing must never go back to an earlier point in the graph.

Running on a distributed infrastructure, the design of DSP applications tries to conveniently exploit different forms of parallelism among the operators. *Task parallelism* is the concurrent execution of different operators on the same or different data. *Data parallelism* is the concurrent execution of multiple instances (replicas) of the same operator on different parts of the stream. The number of parallel instances of an operator determines the operator's replication degree (also known as *parallelization degree*).

In Figure 2.1 we see a DAG used to process a stream of sensor data. There is one data source and six operators with different parallelization degree i.e different number of tasks. Links between operators in the topology indicate how tuples are passed around. Each operator's code is executed by threads (we can assign one task per thread). Data is loaded from sensors and separated by sensor type. The data from the first sensor (A) is

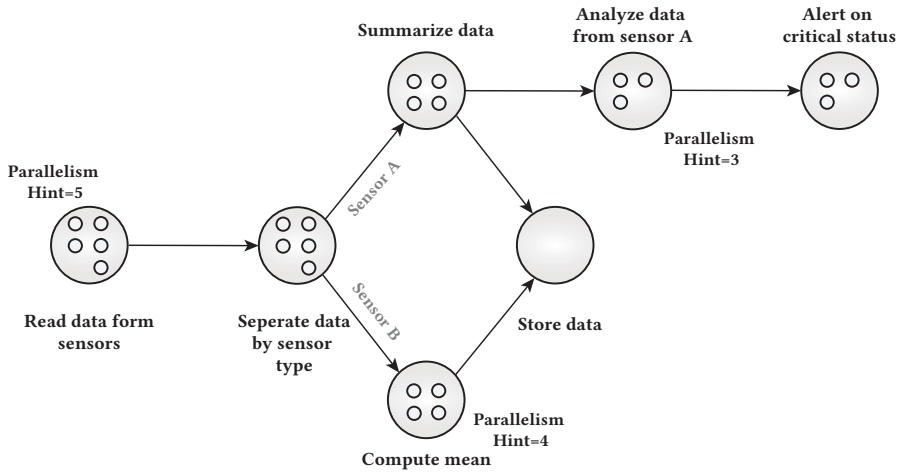


Figure 2.1: A DAG example-"Working with sensor data"

summarized and then analyzed in real-time. In case of a critical value, an alert is sent. Data is also saved for long-term storage and possibly other analysis. Data from sensor B is used to compute a mean value (e.g. per minute) and then is stored in the same store as the data for sensor A.

The dynamic nature of data streams makes the efficient placement of applications' processing tasks on the available cloud resources challenging. Several technologies have emerged specifically to address the challenges of processing high-volume, real-time data. DSPSs take advantage of the inherent characteristics of parallel and distributed computing to meet these challenges.The demand of elaborate orchestration over a collection of machines is a factor of crucial importance in such systems but the their available in-built scheduling techniques are far from optimal. These systems differences and ideal use cases are not yet, clear enough.

DSPSs are designed to handle data streams and manage continuous queries. They execute continuous queries that are not only once performed, but are continuously executed until they are explicitly uninstalled. They produce results as long as new data arrives in the system and data is processed on the fly without the need for storing it. Data is usually stored after processing. Stream processing systems differ from batch processing systems, due to the requirement of real time data processing. The term "real time processing system" refers to a system that responds within "real-world" time deadlines. It guarantees that a certain process will be executed within a given period, maybe a few seconds, depending on the quality of service constraints. The term "real-time" is a bit redundant but many systems use the term to describe themselves as low latency systems.

There are two execution models used in stream processing [22]:

1. **The Stream-Dataflow Approach**, where an application is viewed as a dataflow graph with operators and data dependencies between them (sometimes referred as **operator-based** approach). A task encapsulates the logic of a predefined operator

like filter, window, aggregate or join or even a routine with user-specified logic. A data stream between two operators represents an infinite sequence of data produced by a task, which is available for further consumption. Everything is automatically pipelined.

2. **The Micro-Batch Approach**, that offers a solution to enable processing data streams on batch processing systems. With micro-batching, we can treat a streaming computation as a sequence of transformations on bounded sets by discretizing a distributed data stream into batches, and then scheduling these batches sequentially in a cluster of worker nodes.

In this section, basic features of data stream processing are introduced. In particular, this chapter:

1. highlights the challenges associated with processing streams of big data in Section 2.1
2. provides an overview of the mechanisms that are used to face them in the same section
3. presents basic characteristics of dominant DSPSs mainly in matters of performance and fault tolerance in Section 2.2.1
4. compares these DSPSs based on a list of criteria and presents relevant benchmarks in Section 2.2.2

and from (1)-(4): Informs potential users about the criteria to consider when choosing a framework for a specific job.

In the following paragraph, the issues and requirements that stream processing systems have to meet to excel at real-time stream processing applications are presented. The analysis done there relies mostly on Stonebraker's work [7] but presents the mechanisms used nowadays in the big data era to face relevant challenges. The major available solutions, according to a number of surveys [6, 21, 23–28], which satisfy the stream processing requirements discussed in this section, are Apache Spark Streaming [29], Storm [17], Flink [30] and Samza [9]. These solutions are open-source solutions with active community support.

2.1 Issues/Requirements in Big Data Stream Processing

Stonebraker [7] introduced the requirements for real-time processing of data streams to provide high-level guidance of what to look for when evaluating stream processing solutions. The most prevalent of them can characterize the available stream processing frameworks for big data in ways that are presented below and could be further enhanced. A real-time stream processing system has to:

1. **"Process messages in-stream without any requirement to store them to perform any operation or sequence of operations."**
   Performance related requirements like latency and throughput are extremely important in streaming applications. To meet these requirements, processing has to be done without the costly storage operation. MapReduce can handle large datasets but works using permanent storage. Thus, it fails when it comes to real-time data processing, as it is designed to perform batch processing on voluminous amounts of data. *In-memory computing* provides a solution to this problem and is based on using a distributed main memory system to store and process big data in real time. Main memory delivers higher bandwidth and better latency compared to hard disk. Even if the framework uses memory for caching the frequently used data, the whole job execution performance will be improved significantly [31].

   Despite the dropping price of memory, using large amounts of RAM to run everything in-memory can be expensive, so proper mechanisms are needed to handle it in an effective way.

   If streams are not managed carefully, processing delays can become unacceptable and lead to long queues at a processing node, buffer overflows, and memory exhaustion. Multiple stream processing computations should be interleaved on the same machine to reduce the number of needed connections and assure high throughput and increased performance. On the other hand, heavily used machines result in memory thrashing, node failures and increased network congestion. Overloaded and underutilized machines should be avoided as imbalances lead to increased computations and deteriorate system's performance [18]. Moreover, despite the dropping price of memory, using large amounts of RAM to run everything in-memory can be expensive, so proper mechanisms are needed to handle it in an effective way.

   Spark Streaming is a system that arose to provide in-memory computation effectively. It processes all data in-memory, only interacting with the storage layer to initially load the data into memory and at the end to persist the final results. To implement in-memory computations, Spark uses a model called Resilient Distributed Datasets (RDDs), its in-memory abstraction to work with data. Flink also offers in-memory computations. Its core is built on a data flow streaming engine whose fundamental functionality is pipelining i.e. all tasks have to be online simultaneously in order for the data to be able to flow through the tasks regardless of the node, that a task might reside on [25]. Storm runs also in-memory to process big data at in-memory speed. Memory is a resource of crucial importance for a DSMS. Since in-memory computing is not only about storing but also processing big data in real-time, problems like efficient task scheduling is a matter that it has to deal with [31].

2. **"Support a proper querying language with extensible stream oriented primitives."**

Stream processing systems receive input streams from one or more sources and organize the computations into a directed graph of operators either explicitly or implicitly. When organised explicitly, systems are known as compositional. They offer basic building blocks for composing custom operators and topologies. The user has to implement the whole logic himself and the operators are defined as implementations of classes. When organized implicitly, systems are called declarative and developers are provided with high-level languages that are automatically translated by the system into the operator graph [22], [27].

Querying mechanisms are needed to detect events of interest or compute real-time analytics. General purpose languages like Java have been used as programming tools in streaming applications but using low-level programming schemes results in long development cycles and high maintenance costs. Using a stream processing system with no support for SQL like query languages, requires sound knowledge on imperative style programming and distributed systems to effectively utilize it. Support for SQL-like continuous query languages or SQL with streaming extensions can help towards this direction. SQL remains a reliable query language with high performance for real-time analytics but has limitations when dealing with huge amounts of data [32]. Systems adopt SQL-like languages that represent the processing as queries that get repeatedly and continuously evaluated as new data becomes available. New projects like SQLStream [33] and Apache Calcite [34] emerge to execute queries over big data using a set of streaming-specific extensions to standard SQL. Amazon Redshift [35] is built around industry-standard SQL, with added functionality to manage very large datasets and support high-performance analysis and can be combined with DSPSs like Apache Spark or Storm. There are a number of efforts towards providing users the ability to quickly and cost-effectively build real-time analytics dashboards and applications that can continuously process very high volumes of streaming data.

There is a concept of vital importance when it comes to data stream mining. The data set is assumed to be infinite, creating problems in processing. Not all operators can be evaluated over streams. A stream processing engine has to know when to finish an operation on a stream of data and output an answer. Traditional methods have the advantage of knowing the total size of the set. Sampling is a tool that addresses this problem. Windowing is a sampling method defining the scope of an operation and is a heavily used approach in stream processing. An unbounded stream of data is split into finite sets, called windows, based on specified criteria, like time. A window can be conceptualized as an in-memory table where events are added and removed and computations are calculated on each window of events. In [36] and [37] there are detailed analysis over windowing features. There are stream processing frameworks that offer only the basics, like Storm, Samza. Spark Streaming supports only time windows having a size and slide that are multiple of the batch size it creates. On the contrary, Spark Structured Streaming, a new component in Spark 2.0, offers more windowing possibilities.

Flink also offers a wide range of windowing features. Tumbling windows, sliding windows, session windows and global windows are pre-implemented while users can also implement their own windows. The notions of windowing and time, and the windowing semantics used by prominent stream processing frameworks are further discussed in [38].

3. **"Use mechanisms to provide resiliency against stream imperfections including out-of-order or missing data which are commonly present in real-world data streams."**

   Incomplete datasets, destroyed data, the presence of outliers or biases in the training affect the analysis' accuracy. Missing data is a phenomenon that is inevitable in distributed communication environments. A transmission loss due to broken link between sensors or a malfunctioning sensor leads to missing values from the data. A sensor accident can lead to permanent missing values while a temporal disconnection or network delay leads to temporal loss as data may arrive in a short while [39].

   The assessment of data quality demands significant human involvement and expert knowledge but when it comes to large volumes of unstructured data even semi-automated approaches are not practical. Streaming data and real-time processing outweigh data quality, making data quality management more imperative than ever. To the best of our knowledge, the above problems have not been addressed effectively so far, but there is growing interest towards this direction [39], [40].Techniques like outlier detection, dimensionality reduction, cross-validation and bootstrapping are valuable tools in data quality management but until recently, data quality research has primarily focused on structured data, stored in relational databases and file systems.

   Streaming data is also typically not well ordered in time. Event-time ordered data is uncommon in many real-world, distributed input sources. While receiving a stream of an IoT sensor readings for example, some devices might be offline, and send data after some time. Keeping a strictly time-constrained system waiting, is not an appealing solution. Keeping all windows open forever would also consume all available memory. When it comes to real-time processing, there must be a mechanism to allow windows to stay open for a while. Watermarks are such a mechanism. They enable streaming systems to emit timely, correct results when processing out-of-order data. They are used as a heuristic, assuming that all events before a specific time have been observed. This technique is further explored in [37]. Most prominent DSPSs (e.g. Spark Structured Streaming, Storm, Samza, Flink etc.) implement watermarking techniques.

4. **"Ensure that the apps are up and available and the integrity of the data should be maintained at all times despite failures. The system should have**

the capability to efficiently store, access state information and combine it with live streaming."

Network failures, lack of resources, and network software bugs can cause even more problems in large-scale distributed computing. When large sets of such components are working together there is high probability that at least one component may fail at a given time. Almost all stream processing systems in big data industry provide the ability to recover automatically from faults. Several techniques have been developed to recover fast enough so that the normal processing can continue with the minimal effect to the overall performance. We distinguish three main categories for recovery in stream processing; precise recovery, rollback recovery and GAP recovery (the interested reader can refer to [41] for a thorough analysis of fault tolerance in stream processing engines). Precise recovery provides no evidence of a failure afterwards but there is an increase in latency. In rollback recovery the output produced after a failure is "equivalent" to, but not necessarily the same as, the output of an execution without failure. For example, information may be processed more than once when a failure happens. In GAP recovery, the loss of information is expected in favor of reduced recovery time and runtime overhead.

Storm and Samza use upstream backup techniques that provide roll-back recovery. Generally speaking, a stream processing system can use the upstream backup method to avoid any checkpointing overhead e.g. disk I/Os and data structures (checkpointing is also available in both aforementioned systems). When it comes to state, though, Storm requires the user to manually handle recovery of state [42]. Apache Zookeper [43] has to be used to maintain its cluster state while Samza makes state changes fault tolerant by modeling them as an output log (commitlog or changelog) to a Kafka topic [44]. Systems like Spark Streaming that work with batches usually re-execute the necessary computations in case of failures, yielding the same output regardless of them due to RDDs, that can be recomputed deterministically. RDDs are immutable, meaning that no worker node can modify it, it can only process it and output some results. Each RDD can trace its lineage back through its parent RDDs and ultimately to the data on disk [42]. Flink implements different recovery mechanisms; rollback/restart for finite streams and distributed snapshotting for infinite streams. Flink's checkpointing mechanism stores consistent snapshots of the data stream and operators' state. These fault recovery methods lead to different processing guarantees for each system (and provide fault tolerance in different ways) that are further examined in Section 2.2.

5. **"Be able to distribute its processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent."**

Streams do not have preset lifespan, arrive at non predefined rates and are voluminous. If not managed carefully, processing delays can become unacceptable and lead to long queues at a processing node, buffer overflows, and memory exhaustion. The number of stream computations are much more than the number of machines available for processing. Multiple stream processing computations should be interleaved on the same machine to reduce the number of needed connections and assure high throughput and increased performance. On the other hand, if heavily used machines result in memory thrashing, node failures and increased network congestion. Moreover, a possible node failure will bring down most of the application. Overloaded and underutilized machines should be avoided as imbalances deteriorate system's performance.

Running algorithms in a sequential manner is not efficient. Algorithms should run in parallel with streams of data also partitioned to many distributed processing units (task and data parallelism). Moreover, unlike stateless computations, stateful computations cannot simply be replicated on multiple machines with streams of data been processed in e.g. a simple round robin fashion in parallel. Horizontal scaling requires adapting the graph of processing elements, exporting and saving operators' state for replication, fault tolerance and migration [5]. As stream processing queries are often treated as long running that cannot be restarted without incurring a loss of data, and the application's reconfiguration and re-balancing may be time-consuming, the initial task assignment, where processing elements are deployed on available computing resources becomes more critical than in other systems. Processing must be orchestrated carefully over a collection of machines making task scheduling over a set of machines, a challenge to be faced by this doctoral research.

6. **"Have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications"**

The aforementioned rules make no sense alone, unless an application can process high-volumes of streaming data with very low latency and high throughput, to meet the real-time demands. The ability to process large volumes of data on the fly, as soon as they become available, is a fundamental requirement in today's information systems. There is a rapid increase in the number of available stream processing engines. However, each engine defines its own processing model and execution semantics that affect its performance. For instance, Apache Storm uses the dataflow execution model where streams of data are processed tuple by tuple on continuous operators. On the other hand, Spark streaming creates small batches of streaming data and execute them based on its batch processing engine. More details on ways that can enhance the DSP systems performance are examined in the following chapter, regarding available heuristics along with the relevant discussion.

Taking into account the aforementioned requirements, the nature of streams poses several processing challenges. The matters of performance and message processing assurance seem to rise to the top when choosing a framework for a specific application. In the following sections, we are going to further examine the aforementioned available DSPSs with respect mainly to these matters to further check and compare their capabalities, and the results of their built-in mechanisms.

2.2 DSP Frameworks Assessment

Research and developments in big data stream processing systems are ongoing and of great interest against the challenges posed by business trends.The available frameworks have several differences in their architecture and in the processing model they use. Choosing a system that can guarantee fault-tolerant, high performance stream processing based on user's needs is quite cumbersome. To support such a decision, brief reviews over initial solutions [23, 31, 45] and comparisons based on the basic selection criteria such as language support and documentation [24] soon led to full surveys over a number of available choices [5, 21, 25, 27, 46]. Most of these works focus on metrics like latency, throughput and message processing assurance and the need for benchmarks to support potential users is usually praised. Details on aspects like memory and resource management are also usually thoroughly discussed [25] just like the need for research on scheduling of streaming tasks to support aspects like fault tolerance and high performance of the execution engine [27].

To select the major solutions and perform a comparison between them, several surveys ([5, 21, 23–25, 27, 31, 46] were consulted. We selected a subset of the state of the art systems, able to satisfy the stream processing requirements mentioned in the previous section, that use the most representative mechanisms to do so. Finally, open-source solutions with active community support were selected to enable working with scheduling practices in the following chapters. The mechanisms overviewed to handle the stream processing requirements provide different functionality characteristics to each DSPS.

In this section, the functionality of a streaming engine is divided mainly in terms of performance and fault tolerance to permit the evaluation of the selected DSPSs. The aforementioned related surveys were also helpful in defining the list of criteria used for the purposes of this study. This chapter also presents experiments conducted, to investigate the performance and the fault tolerance mechanisms used.

2.2.1 Evaluation and Comparison of Stream Processing Frameworks

All steaming solutions examined in this section use a parallel and distributed architecture that allows portioning of data streams and parallelisation across a cluster of machines. The attributes that are also examined are the following:

Processing Model: The selection of a processing model for a system varies from batch processing and micro-batch processing to stream processing tuple by tuple. Batch processing systems such as MapReduce are beyond our interest.

Stream Primitive: Refers to the main data structure in a streaming system. These systems use various words for such concepts.

Latency: Refers to the elapsed time from job submission to receiving the first response.

State Management: Streaming computations can be either stateless or stateful. "A stateless program looks at each individual event and creates some output based on that last event". For example a streaming program might receive traffic data and raise an alert in the event of traffic light violations. "A stateful program creates output based on multiple events taken together" [47]. For example, creating an alert after receiving two traffic light violations that differ by less than 5 minutes is a stateful computation. The frameworks examined use various strategies to store state or may not store state at all.

Throughput: Refers to the average number of jobs or tasks or operations performed per time unit.

Delivery Guarantee: Refers to the "level of correctness" of the results produced after a failure and a successful recovery of the system compared to what the results would be without any failures. Stream processing systems are characterized by the following three semantics [9]:

- *at-most-once delivery*, which drops messages in case they are not processed correctly, or in case the processing unit fails. This is usually the least desirable outcome as messages may be lost.
- *at-least-once delivery*, which tracks whether each input was successfully processed within a preset timeout. It this way, it guarantees that messages are redelivered and re-processed after a failure. If a task fails, no messages are lost, but some messages may be redelivered. In case the effect of a message on state is idempotent, no problem occurs if the same message is processed more than once. A duplicate update will not change the result. However, for non-idempotent operations such as counting, at-least-once delivery guarantees can give incorrect results. This approach is good enough for many use cases but it may cause duplicates.
- *exactly-once semantics*, which uses the same failure detection mechanism as the at-least-once mode. Messages are actually processed at least once, but duplicates can be avoided via various techniques. Such systems guarantee that the final result will be exactly the same as it would be in the failure-free scenario. This is the most desirable feature but it is difficult to guarantee in all cases.

Table 2.1: Classification of Streaming Solutions

Platform/ Criteria	Processing Framework	Stream Primitive	Latency	Throughput	Stateful Operations	Guarantee	Programming Model	API languages	Contributers (Github) (10/2020)
Storm	Streaming	Tuple	Subsecs	Low	No	At least once (exactly with Trident)	Compositional	Any	339
Spark Streaming	Micro-Batch	Dstream	Few Secs	High	Yes	Exactly once	Declarative	Java, Scala, R, Python	1565
Samza	Streaming	Message	Subsecs	High	Yes	At least once	Compositional	JVM languages	122
Flink	Hybrid	DataStream	Subsecs	High	Yes	Exactly once	Declarative	Java, Scala, Python	765

Programming Model: Systems can be either compositional when users have to model the streaming application as graph explicitly or declarative when users are provided with higher level abstractions.

API languages: Refers to the languages that someone can be used to develop an application for this framework.

Contributers: Refers to the respective community of contributors based on Github.

Table 2.1 summarizes how four different frameworks support the above features.

Storm implements the dataflow model. The topology of an application is described by using operators (or components), named "spouts" and "bolts", referring to data sources and elements that process data in the form of tuples respectively [48]. Since a tuple is processed as it arrives, Storm has sub-second latency. Its mechanism supports low throughput mainly because of its acknowledge mechanism. Each record that is processed from an operator sends an acknowledgement back to the previous operator, indicating that it has been processed. This mechanism may also falsely classify a number of records as non-aknowledged. Therefore, these records will have to be re-processed leading to even lower throughput.

However, it is based on the 'fail fast, auto restart' approach that allows it to restart the process once a node fails without disturbing the entire operation. This feature makes Storm a fault-tolerant engine. To provide guaranteed delivery, it uses the upstream backup mechanism along with record acknowledgements [17]. In case of failure (e.g. worker failure), if not all acknowledgements have been received, the records are replayed by the spout [49]. In this way, no data loss will occur but duplicate records may pass through the system. Mutable states may be incorrectly updated twice. That's why it offers at-least-once delivery guarantee. In some cases, dropping data is not a problem so users can disable the fault tolerance mechanism by setting the number of acker bolts to 0. Micro-batch processing is offered by Trident Storm for higher throughput. In Storm Trident the state can be managed automatically, so it does guarantee state consistency (exactly-once delivery) but state is kept in a replicated database which is expensive, as

all updates are replicated across the network [42]. Apache Storm is a compositional system that expects user to explicitly define the application DAG.

Spark Streaming batches up events within a short frame before processing arrived data and is the most active project in terms of community numbers (mainly because of the batch processing capabilities of Apache Spark). At the low-level, data is represented as RDDs and computations on these RDDs can be represented as either transformations or actions. The abstraction for data streams is called Dstream and it consists of an RDD sequence, containing data of a certain stream interval. DStreams let users apply transformations to them.

Streaming computations in Spark Streaming represent a series of batch computations of a definable time interval size [50]. Adapting the batch methodology for stream processing involves buffering the data as it enters the system leading to few seconds latency. There is a penalty of latency equal to the micro-batch duration. Waiting to flush the buffer also leads to an increase in latency. Nevertheless, the buffer allows Spark Streaming to handle a high volume of incoming data, increasing overall throughput [46]. As we can see there is a trade-off between low latency and high throughput. Systems based on micro-batching can achieve high throughput but in case processing of a batch takes longer in downstream operations than in the batching operations, the micro-batch will take longer than configured. This may lead to more and more batches queueing up (or to a growing mini-batch size) [44]. While Spark Streaming may be adequate for many projects, it is not a true real-time system. Zaharia et al. [51] mention, though, that while this model is slower than true streaming, the latency can be minimized enough for most real-world projects due to the use of RDDs. RDDs allow in-memory computations in a fault-tolerant manner, avoiding writing outputs to replicated, disk storage systems, yielding to time consuming disk I/Os. Spark Streaming provides support for both stateful and stateless computations and guarantees that batch level processing will be executed in an exactly once manner. This is achieved by tracking the lineages in each DStream. All state in Spark Streaming is stored in RDDs [51]. It is a declarative system as it introduces several abstractions for representing data and managing different types of computations. The code defines just the functions that need to be performed on the data and Spark implies the corresponding DAG from the functions called.

Samza is a stream-processing framework based on the Publish/Subscribe model. It listens to a data stream, processes messages which are its stream primitive as they arrive, one at a time, and outputs its result to another stream. It is tightly tied to the Apache Kafka messaging system [44] for streaming data between tasks. Kafka offers replicated storage of data that can be accessed with low latency, so Samza jobs can have latency in the low milliseconds when running with it. Samza allows tasks to maintain state by storing it on disk (typically using Kafka). This state is stored on the same machine as the processing task, to avoid performance problems. By co-locating storage and processing on the same machine, Samza is able to achieve high throughput [9]. It also tracks whether a message is delivered or not and it redelivers it in case of failure, to avoid data loss using a checkpointing system but can only deliver at least once guarantees. If a Samza task

fails and is restarted, it may double-count some messages that may have been consumed since the last checkpoint was written. The topology of a Samza job is explicitly defined by the user's code [27].

Flink is an hybrid solution [52]. The need to manage different workloads under a coherent architecture led to several design patterns with the most popular being the "Lambda Architecture". The complexity of using different batch and streaming architectures paved the way to the "Kappa architectural" pattern that fuses the batch and stream layers together. Flink is a materialization of the Kappa archtecture. Despite the fact that it relies on a streaming execution model, it is possible to process both bounded and unbounded data, with two APIs running on the same distributed streaming execution.

The basic data abstraction for stream processing is called DataStream. It executes arbitrary dataflow programs in a data-parallel and pipelined manner, which results in achieving low latency. Apache Flink's dataflow programming model provides event-at-a-time processing. Tuples can be collected in buffers with an adjustable timeout before they are sent to the next operator to turn the knob between throughput and latency. It performs at large scale, running on thousands of nodes with very good throughput and latency characteristics based on existing benchmarks. When using stateful computations, it ensures exactly once semantics. Apache Flink includes a lightweight fault tolerance mechanism based on distributed checkpoints. Its algorithm periodically draws consistent snapshots of the current state of the distributed system without missing information and without recording duplicates. These snapshots are stored to a durable storage. In case of failure the latest snapshot is restored, the stream source is rewinded to the point when the snapshot was taken, and is replayed [30]. Flink is a declarative system, providing higher level abstractions to users like Spark. The DAG is implied by the ordering of the transformations while its engine can reorder the transformations if needed.

2.2.2 Comparison Results

The provided overview reveals that there is no system able to do everything and no system that does nothing. All systems do something but they do it differently. Benchmarking can be a good way to compare them, especially when it has been done by third parties. To shed some light on the performance of the above engines, a few experiments that investigate latency, throughput and the impact of their fault tolerance mechanism have been conducted. Some examples are provided below:

- Cordova [49] compared Spark Streaming and Storm Trident. He provided a benchmark of both systems over different tasks and processing tuples of different sizes. The main conclusion was that Storm Trident is around 40% faster than Spark, processing tuples of small size. However, as the tuple's size increased, Spark has better performance maintaining the processing times.
- Chintapalli et al. [53] designed and implemented a real-world streaming benchmark focusing on Storm, Flink and Spark Streaming and found that Storm

and Flink have much lower latency than Spark Streaming at fairly high throughput. On the other hand, Spark Streaming is able to handle higher throughput and its performance is quite sensitive to the batch duration setting.

- Perera et al. [54] compared Flink with Spark against two benchmarks, Intel HiBench Streaming and Yahoo Stream. Both systems performed similarly under different loads but Flink demonstrated a slightly better performance at lower event rates, mainly because of Spark Streaming's micro-batch technique. The CPU utilization was similar in both systems but when it came to memory usage, Flink needed less amount of memory than Spark.

- Lu et al. [55] also proposed a benchmark definition named StreamBench. They applied StreamBench to Spark Streaming and Storm. They found that Spark tends to have larger throughput (even about 5 times that of Storm's) and less node failure impact compared to Storm. Storm, though, has much lower latency (even 50 times less) than Spark, except with complex workloads under large data scale, for which its latency may be multiple times of Spark's.

- Karimov et al. [56] proposed a benchmark framework and conducted experiments with Storm, Spark and Flink. Both Flink and Spark are robust to fluctuations in the data arrival rate in aggregation workloads while for join queries, Flink behaves better. In case latency is a priority, Flink seems to be the best choice and has better overall throughput when tested in aggregation and join queries.

To provide accurate and customized recommendations, we can't completely rely on benchmarking in stream processing as even a small change in configuration or use case can completely change the numbers. Moreover, the above technologies may have established themselves as leaders but they have strong supporting communities and are evolving very fast. Consequently, it is not easy to make a clear ranking with quantifiable results. Combining the above overview (Table 2.1) with available benchmarks, though, some conclusions, arise:

- Storm works with very low latency but can deliver duplicates and cannot guarantee ordering by default configuration. Since it does not provide implicit support for state management, it does not fit in cases of complex event processing. Nevertheless, it excels in case of non-complicated streaming use cases, where latency is of crucial importance, due to its true streaming nature and maturity.

- Samza has to be integrated tightly with Kafka and YARN to provide high performance, flexibility, and state management. It is not easy to be used without them in the processing pipeline, while deploying such a system would require extensive testing to make sure that the topology is correct. Nevertheless, in case these technologies are already incorporated, it is a mature, fault tolerant solution providing high performance (both in matters of latency and throughput).

- Spark is very popular, mature and widely adopted with a strong community supporting it. It provides high throughput, it is fault tolerant because of its micro-batch nature providing exactly one guarantee, and can be used in case sub-latency is not required. Nevertheless, it lags behind Flink in many advanced features. Its new release (Structured Streaming) is equipped with several good features and promises to yield subsecond latency, but at the time of writing, it's early for this ambition to be achieved.

- Flink provides true stream processing with batch processing support. It is heavily optimized, incorporating several innovations like light weighted snapshots and it seems that it's the leader in the DSMS landscape. As we saw, most of the wanted aspects (low latency, high throughput, exactly once guarantee, state management) are provided. Nevertheless, there was lack of adoption initially and its maturity is a matter of concern. Its community support is also smaller than Spark's. However, this seems to change rapidly.

Consequently, the best possible answer provided to users that want to learn which is the best possible solution is that it depends on their needs. Future considerations should also be taken into account. For instance, in simple cases Storm may seem a good idea. Nevertheless, in case advanced requirements involving complex event processing like aggregations and joins occur later, or batch-oriented tasks should also be implemented, advanced streaming frameworks like Spark Streaming or Flink should be preferred. Changes in existent infrastructure and re-training employees may lead to huge costs in time and money.

Finally, we should keep in mind that coding in declarative systems is much easier than in compositional as users are provided with higher level abstractions and imply the DAG through their coding. Optimizations can be done by the system. Nevertheless, in compositional systems the code is at the complete control of the developer. If there is a need for fast and easy implementation, systems like Spark or Flink should be preferred but if complete control over the application's graph is needed, then Storm or Samza should be chosen.

As noticed, the best fit for each situation depends upon several factors. Understanding the mechanisms and characteristics of the aforementioned architectures makes it easier to pick or at least filter down the available options. Work-in-progress evaluation can then support users to make the best possible choice based on their needs. Changes in configuration, tuning or available infrastructure can alter the results and lead to severe improvements. Scheduling strongly affects the performance and fault tolerance in a stream processing system [27]. Most streaming systems allow the user to specify custom scheduling for tasks and the interest on this aspect grows widely. This led to the investigation of the task placement and scheduling problem in a distributed setup. In the next chapter, relevant efforts are reviewed and this thesis is positioned with respect to the state of the art.

Chapter 3

Related work

"Curiosity, especially intellectual inquisitiveness, is what separates the truly alive from those who are merely going through the motions."-Tom Robbins

A number of DSPSs has been developed in academic, open source, and industry communities. Initially, Section 3.1 focuses on the scheduling techniques of the systems discussed in the previous chapter. Then Section 3.2 presents different scheduling approaches found in the literature, that can be incorporated in these systems. In particular, this chapter:

1. presents an up-to-date overview and classification of scheduling policies for DSPSs (Section 3.2) and positions this work with respect to the state of the art in Section 3.2.2.

2. detects the factors that affect scheduling decisions in the discussion of Section 3.3.

and based on (1)-(2) informs the potential user about the matters to consider when designing an effective scheduling technique for a DSPS.

3.1 Data Stream Processing Systems (DSPS)

The historical evolution of stream processing frameworks is described in [57]. Traditional Database Management Systems (DBMSs), that were built around a persistent storage to optimize user-triggered queries, evolved to Data Stream Management Systems (DSMSs) that execute continuous queries to provide updated answers as soon as new data arrives. DSMSs, also offer an SQL-like declarative language to define continuous queries. As an example, Aurora [58] was an early implementation of a DSMS, that was used to parallelize streaming computations including rich operation and windowing semantics. Initially, it was designed as a single site stream-processing engine. Its predecessor, Borealis [59] was a stream processing engine that focused on balancing load on individual machines and distributing load shedding in static environments. Borealis was based on ROD (resilient operator distribution) to determine the best operator

distribution plan, trying to be closest to an "ideal" feasible set, having a maximum set of machines underloaded.

At the aftermath of Big Data and the IoT, new challenges were posed to traditional stream processing engines. These challenges arose from the need to work with huge amount of data, requiring massive parallel stream processing capabilities. Data stream processing systems (DSPSs) were implemented to do for real-time processing, what Hadoop did for batch processing, using in-built scheduling techniques.

As mentioned in Chapter 2, the major available DSPSs are Apache Spark Streaming, Storm (and its descendant, Heron), Flink and Samza [6]. These solutions are open-source with active community support but there are also commercial solutions like Amazon Kinesis [60], and IBM InfoSphere Streams [61]. Spark Streaming and Storm are the two main representatives of the available execution models for processing streams.

Apache Spark Streaming [29] batches up events within a short frame before processing the arrived data, offering full in-memory computations. Jobs are applied on DStreams according to users' application's operators. Each job is portioned into tasks and a Spark's scheduler is responsible for their assignment to available resources. Its scheduler runs jobs in FIFO fashion and each application tries to use all available nodes. Although job dependencies can be captured with FIFO, this approach can result in increased latency when a long running job delays jobs behind it [6].

Apache Storm [17] follows the operator-based model, as it processes a tuple as it arrives. It uses a round-robin strategy to assign tasks to nodes' slots equally. In this way, logical links between tasks are not taken into consideration and the inter-node communication costs may increase. This simplistic scheduling method frequently leads to low efficiency in load balancing among the available worker nodes.

Apache Flink [30] can support the same capabilities as Spark but works as a native stream engine and handles several challenges better than Spark in stream processing (e.g. in case of recovery and latency). It also seems to have more capabilities than Storm. Nevertheless, Storm has a larger community and as it is a mature project, there is extensive literature on scheduling techniques on it. Apache Flink relies on a streaming execution model but can process both bounded and unbounded data, with two APIs running on the same distributed streaming execution. Its core is built on a data flow streaming engine, whose fundamental functionality is pipelining. A slot in a machine runs one pipeline which consists of multiple successive (communicating) tasks. Its system defines which tasks may share a slot and which tasks must be strictly placed into the same slot. It employs a schedule-once, long-running allocation of tasks and uses an immediate scheduling and a queued scheduling algorithm that work in an arbitrary fashion. The first one returns a slot immediately when there is a request, while the second one queues the request and returns the slot, whenever it is available.

Apache Samza [9] provides a unified programming API for both batch and stream processing. It is based on the publish/subscribe model that listens to a data stream and processes messages (tuples in Storm) as they arrive, one at a time. It is tightly tied to the Apache Kafka messaging system [44] for streaming data between tasks and Apache YARN [62] the for distribution of tasks among nodes in a cluster. YARN is configured to use a fair scheduler with continuous-scheduling enabled.

The aforementioned DSPSs are presented in the corresponding categories of Table 3.1. All systems provide an abstraction layer where to execute DSP applications and allow their users to focus solely on the application logic. Tasks related to the application placement, distribution, and execution are managed by the frameworks themselves but the approaches incorporated in these systems are not optimal. They do not take into consideration the application's structure that is about to be executed or prior knowledge of the cluster's condition. However, these systems allow users to specify custom scheduling for tasks.

3.2 Heuristic Scheduling Approaches

Several heuristic algorithms, that attempt to choose the optimal scheduling technique to maximize their performance have been proposed. Stream processing systems following the micro-batch approach have several advantages over stream processing systems that process data by one record at a time, like fast recovery from failures, better load balancing and scalability. Micro-batch systems are optimized for throughput but have increased query response time, since each input has to wait until a batch is formed. Extremely small batches could possibly minimize this extra latency, but this would cost extra overhead. On the other hand, operator-based stream processing works better when we have to deal with strict real-time constraints but is prone to faults [63]. To provide the necessary system performance at high load incoming data, efficient scheduling of streaming applications with additional processing mechanisms are needed.

Scheduling decisions are devided in *offline* (using static algorithms) and *online* (using dynamic algorithms). Algorithms may rely on predefined characteristics of streaming topologies (offline) or may gather information by monitoring systems (online). Offline decisions rely on the knowledge a scheduler has before any task is placed and running. Online decisions refer to information gathered during the execution of user's application, after the initial placement of its tasks over the cluster's nodes. We are going to further examine and categorize works proposing heuristic algorithms (or sometimes even whole architectures) that try to choose the optimal placement of resources and task executors to maximize their performance.

In the following sections, heuristics regarding the scheduling problem in DSPSs found in literature are reviewed. Most of the heuristics are implemented based on Apache Storm. Table 3.1 presents a classification of the prominent DSPSs described above and heuristic approaches found in literature, grouped by their execution model and the type of scheduling decisions they make.

3.2.1 Static Approaches Using Mini-batches

Spark Streaming belongs in this category. In most static scheduling approaches over systems that use micro-batches, the matter of batch-sizing is a factor of crucial importance. Such systems do not deal the way the systems places the data but with the data itself. Batch-sizing affects latency and can facilitate the scheduling and possible

Table 3.1: An overview of scheduling approaches in stream processing big data frameworks.

System	Execution Model	Scheduling Decisions	Awareness	Based on	DSPS used
Spark Streaming [29]	Micro-batches	Offline	-	FIFO	
Das et al. [64]	Micro-batches	Offline	Resources Workload	Control module, Fixed-point Iteration Optimization Technique	Spark
Liao et al. [65]	Micro-batches	Offline	Workload	Timer	Spark
Drizzle [66]	Micro-batches	Offline	Topology, Performance	TCP congestion control, AIMD policy queues	Spark
Storm [17]	Operator	Offline	-	Round Robin	
Samza [9]	Operator	Offline	-	Fair scheduling	
Eidenbenz and Locher [13]	Operator	Offline	Topology, Resources	Graph Theory	-
Aniello et al. [67]	Operator	Offline	Topology	Round-robin	Storm
R-Storm [19]	Operator	Offline	Topology, Resources, Resources Requirements	BFS, Euclidean distance, Linear Programming	Storm
RB-storm [68]	Operator	Offline	Resources, Resources Requirements	Resource Imbalance Vector, Linear Programming	Storm
GA Storm [69]	Operator	Offline	Topology, Performance	A priori knowledge, Genetic algorithm, Linear Programming	Storm
P-Scheduler [70]	Operator	Offline	Topology, Traffic, Workload	A priori knowledge, Graph partitioning	Storm
I-Scheduler [71]	Operator	Offline	Topology, Resources, Traffic, Workload	A priori knowledge, Graph partitioning	Storm
Shukla and Simmhan [72]	Operator	Offline	Topology, Resources, Workload, Performance	A priori knowledge, Linear Programming, Queueing theory	Storm
Rychly et al. [12]	Operator	Offline	Resources, Performance	A priori knowledge	Storm
Cardellini et al. [73]	Operator	Offline	Topology, Resources, Resources Requirements, QoS attributes	Linear Programming	Storm
MT-scheduler [74]	Operator	Offline	Topology, Resources, Resources Requirements	Dynamic Programming, Linear Programming	Storm
Janßen et al. [75]	Operator	Offline	Topology, Resources, Resources Requirements,	Linear Programming	Flink
Bframework [76]	Operator	Offline	Topology, Resources, Workload	Linear Programming, Queries attributes	Storm
Tantalaki et al. [77]	Operator	Offline	Topology	Linear Algebra, Matrix transformations	Storm

Table 3.2: (Continued)

System	Execution Model	Scheduling Decisions	Awareness	Based on	DSPS used
Flink [30]	Operator based(Hybrid)	Online	Resources	Task locality, Immediate and Queued scheduling	
A-Scheduler [78]	Micro-batches	Online	Topology, Performance	FIFO, FAIR, Reinforcement Learning, Monitoring	Spark
Lever [79]	Micro-batches	Online	Resources, Performance	Iterative Learning Control model, Monitoring	Spark
DRS [63]	Operator based	Online	Resources, Performance	Erlang queuing model, Jackson network, Linear Programming Monitoring	Storm
Aniello et al. (Dynamic) [67]	Operator based	Online	Workload Traffic	Linear Programming Monitoring	Storm
T-Storm [80]	Operator based	Online	Workload, Traffic	Machine Learning, Linear Programming, Monitoring	Storm
Meng-meng et al. [81]	Operator based	Online	Topology Workload Traffic	Matrix model, A priori knowledge Monitoring	-
E-stream [82]	Operator based	Online	Topology, Resources, Data stream rate	EFTime first strategy (priorities) Linear Programming Monitoring	Storm
Dhalion [83]	Operator based	Online	Workload Performance, SLO	Linear Programming Monitoring	Heron
Safaei [84]	Operator based	Online	Topology Resources Performance Data stream rate	System's response time function, First-Fit, EDF, Linear Programming Monitoring	Storm

rescheduling of tasks and data. Attempts like [64] and [65] rely on dynamically adjusting batch intervals, according to the number of events arrived in current interval. For example, reducing the interval near data peaks proved to be a good choice to smooth the overall delay time. Their approaches are implemented on the existing scheduling framework of Spark Streaming.

Towards this direction, choosing an appropriate group size of mini-batches is important to ensure that *Drizzle* [66] achieves the desired performance. This topology-aware strategy has as main goal to decouple processing intervals from coordination intervals. Assuming the data sources remain the same, and the cluster configuration is static, the same task-to-worker mapping can be used for every micro-batch. An adaptive group-size tuning algorithm inspired by TCP congestion control is used to combine multiple micro-batches into a single message. During the execution of a group, counters are used to track the amount of time spent in various parts of the system and a policy analogous to AIMD determines the coordination frequency for a job. Query optimization techniques are used to achieve better throughput. Drizzle is implemented by extending Apache Spark to achieve lower latency and faster failure recovery.

3.2.2 Static Approaches in Operator-based Systems

Apache Storm and *Samza* belong in this category. Most of the heuristics found in literature are based on Storm. Eidenbenz and Locher [13] established a theoretical foundation for the task allocation problem in systems like Apache Storm. They used a fixed set of resources with uniform capacities and bandwidth and stream topologies are expressed as directed serial parallel decomposable graphs.In their work they also prove that the task allocation problem is NP-hard.

Aniello et al. [67] proposed a topology-aware scheduler for Storm. Their approach identifies possible sets of operators' threads to be scheduled on the same slot by looking how components are interconnected within the topology (DAG). Finally slots are assigned to nodes in a round robin fashion. Their approach tries to balance the total CPU demand of each worker. As load imbalances are possible due to workload fluctuations, monitoring is used by their dynamic adaptive scheduler to handle these cases, but this is described in Section 3.2.4. This approach improves system's performance by reducing the inter-node communication cost and the required buffer space (for each task and thus for each worker node).

Peng et al. [19] implemented *R-Storm*, a topology and resource-aware scheduling approach that also schedules adjacent components' tasks as close as possible to reduce communication latency using breadth first traversal (BFS). It also tries to maximize the resource usage in a slot to minimize resource waste in nodes, using a resource-aware distance function. R-Storm's scheduler yields better performance than the default round-robin Storm's scheduler but it cannot control the performance when CPU sharing occurs. To minimize the resources wastage, De Xiang et al. [68] implemented a scheduling algorithm with the consideration of the worker nodes' load, named RB-storm. To do so, they applied a Resource Imbalance Vector (RIV) to represent the imbalance of resource utilization in tasks and worker nodes. Both Peng et al. [19] and De Xiang et al. [68]

worked with homogeneous clusters and considered memory as a constraint in their analysis. Resource waste is minimized with respect to the knowledge about the node capacities and the task requirements but this information has to be provided by the user.

Smirnov et al. [69] proposed another topology-aware strategy that also takes into consideration system's performance. Their approach is based on a genetic algorithm (GA) and uses performance models of executors on specified nodes to estimate their throughput. In their experiments, they allowed CPU-sharing between tasks and they proved that maximum tasks' performance can be achieved via minimum CPU sharing, consuming though the maximum number of cores. In their experiments, GA scheduler was better in handling the high workload in several topologies, whereas R-Storm's and Storm's performance remained almost equal and low. Their performance models demand either history data or data collected during runtime.

Taking advantage of prior knowledge, Eskandari et al. [70] presented *P-scheduler*. This scheduler uses data transfer rates between tasks and topology workload obtained by running the known topology a priori. It places highly-communicating task pairs to the same working node applying hierarchical graph partitioning. Their work assumes that the cluster is homogeneous. Later, they extended their work [71] in heterogeneous clusters and proposed *I-Scheduler*, an iterative graph partitioning-based heuristic algorithm. This approach finds partitions of highly communicating tasks, sized according to node capacities and fuses each partition into a single task. A node's capacity is defined as the sum of the CPU speed for all cores within a node. While these schedulers can estimate the necessary number of nodes for an application and maximize resource utilization, memory resources and their consumption are not taken into account.

Shukla and Simmhan [72] also utilized a priori knowledge of the tasks' performance (using micro-benchmarks) to get a predictable scheduling behavior given a fixed input stream rate. Apart from assigning threads to the working nodes to ensure an expected performance, they also examined the matter of allocation of threads and resources for a DAG. This means that their scheduler determines the appropriate number of replicas per task and quanta of computing resources. Their analysis is based on CPU and memory resources and offer lower resource requirements and VM cost compared to R-Storm. Benchmarking of application on a particular cluster prior to its run in production was also used by Rychly's et al. [12] resource and performance aware strategy. Their scheduling algorithm works on heterogeneous clusters and employs design-time knowledge (a tagging process is required) and benchmarking to take decisions.

In several cases, the allocation problem is considered as a linear programming problem (e.g. [19, 68, 69, 72]). Taking advantage of linear programming privilleges, Cardellini et al. [73] provided a solution that takes into consideration the heterogeneity of computing and network resources to optimize different QoS requirements. Their proposed formulation considers user-oriented QoS attributes like end-to-end latency and application availability, and network-related attributes like network usage, inter-node traffic, and elastic energy. Memory consumption is not considered in their QoS metrics. Workload balancing and distribution are also not considered.

Towards this direction, Al-Sinayyid and Zhu [74] proposed *MT-scheduler* that uses a dynamic programming technique to efficiently map a DAG onto heterogeneous systems.

They proposed a polynomial-time heuristic solution that is based on computing and data transfer requirements, and the capacity of the underlying cluster resources. Memory is not considered in node attributes in this work. Their system performance optimization is realized by estimating and minimizing the time incurred at the computing and transfer bottlenecks. To avoid system overloading, the scheduler is called periodically to update the mapping process.

Janßen et al. [75] also assumed a static environment with heterogeneous resources. In their work, QoS metrics like response time, bandwidth congestion, and resource fitting are combined into an optimization function to implement metaheuristic methods for near-optimal task placements. Unlike most approaches that work with Apache Storm, they extended Apache Flink's scheduling workflow.

Mortazavi-Dehkordi & Zamanifar [76] combined linear programming with queries' attributes. They proposed *Bframework* that examines the topology structure of a query to estimate the size of output stream flow of its operators to profile and partition them. Different scheduling strategies are applied to each partition. At first, operators are assigned to thread computing units and the identified threads are assigned to processes. Their offline scheduler (they also extended their work to dynamic environments) finally assigns the processes to a set of available computing nodes. The aim is to minimize the inter-operator traffic load and thus the tuple latency of the accepted queries, distribute and balance the operators' workload. The complexity of their solution is logarithmic and memory is not considered in cluster resources.

This work belongs in this category. It presents a static and topology-aware formulation that represents the task allocation and scheduling problem. The operator-based execution model and Apache Storm's semantics are used as in most of the systems mentioned in the literature. DAGs are used to represent streaming applications. The developed policy uses linear algebra and matrix transformations for all the necessary processes, while linear programming seems to be the dominant strategy in most of the aforementioned solutions.

Most of the static topology and resource-aware policies try to improve the system's performance by considering the topology's structure and the capability of the resources, but they generally ignore the resource load. The policy in this thesis improves the system's performance using an algorithm of linear complexity on a given topology's structure, that takes advantage of a pipeline-based strategy to reduce the required buffer space. Also most of the state of the art strategies ignore the memory consumption issue..

Traffic and workload-aware policies discussed above demand either prior knowledge to improve system's performance or users to provide design time information. However, users usually ignore the application's run-time resource demands. Of course, dynamic approaches can adapt to run-time needs but this requires monitoring and re-scheduling as discussed in the sections below.

3.2.3 Dynamic Approaches Using Mini-batches

Generic scheduling solutions for provisioning to applications competing for cloud resources in most of the aforementioned systems assume that users know the amount

of resources their application needs, and how to distribute these resources internally. Dynamic solutions try to resolve this and make systems adaptive based on the cluster's conditions. To face the dynamism inherent in streaming workloads using mini-batches, Cheng et al. [78] proposed *A-scheduler*. It is an adaptive scheduling approach that dynamically schedules multiple jobs concurrently using different policies based on their dependencies. The data dependency between jobs is identified by profiling the DAG of an application while accepting a job submission. A resource allocator applies Fair scheduling for independent jobs and FIFO for dependent ones. It also collects performance statistics like end-to-end latency for each job and system throughput to automatically adjust the job parallelism settings and resource sharing policies. The tuning problem was formulated as a reinforcement learning process that uses a performance-aware approach. A-scheduler was implemented in Spark.

Chen et al. used pre-scheduling to design *Lever* [79]. The authors focus on the straggler problem; re-scheduling stragglers during the task execution period, increases the processing time of the micro-batches and causes expensive data relocation, as the data has already been dispatched. However, batched stream processing jobs are usually recurring with predictable characteristics. Lever monitors and periodically collects and analyzes the historical job profiles of the recurring micro-batch jobs. It then predicts stragglers using an Iterative Learning Control (ILC) model, which is designed to track control of the systems working in a repetitive mode, to estimate node capacities. Finally, suitable helper nodes are chosen. This approach was implemented as an extension of Spark Streaming.

3.2.4 Dynamic Approaches in Operator-based Systems

Fu et al. [63] designed and implemented *DRS*, a dynamic, operator-based, resource-aware scheduler. Their greedy algorithm takes into account the number of operators in an application and the maximum number of available processors that can be allocated to them, and tries to find an optimal assignment of processors that results in the minimum expected total sojourn time. They estimated the total sojourn time of an input by modeling the system as an open queuing network (OQN). The performance model is built based on a combination of one of Erlang's models and the Jackson network. The system monitors the actual total sojourn time and checks if the performance falls, or if the system can fulfil the constraint with less resources and reschedules if necessary. DRS was integrated into Apache Storm.

As already discussed, Storm's scheduler apart from using all the available worker nodes, it also does not consider links between tasks that are about to be hosted in these nodes. Inter-node and inter-process traffic are factors that make a significant impact on system's performance. *Aniello's et al.* [67] second scheduler and *T-Storm* [80] are approaches that take into consideration the communication patterns of the application to reduce the inter-node and inter-slot traffic in the cluster by assigning tasks that communicate with each other to the same node or adjacent nodes. Workload and traffic load information are collected at run-time. *T-Storm* can also estimate future load using a machine learning prediction method. Both approaches adapt the initial allocation

of executors to the evolution of the load and use Apache Storm for their approaches evaluation. *Meng-meng et al*. [81] obtain the topology by recording workload of nodes and communication traffic through switches a priori. A matrix model is used to describe the real-time task scheduling problem. They evaluated their algorithm by deploying their own stream processing platform SpeedStream and comparing to Apache Storm's and S4's algorithms.

Elasticity is a matter of crucial importance in online environments to determine how to scale for data stream fluctuating with time and schedule resources according to the current arrival rate of a stream. Dawei et al. [82] proposed an elastic online scheduling framework (*E-stream*), that works with multiple DAGs, using the available capacity of computing nodes and the input rate of data stream. E-stream quantifies computation and communication cost, relationships between the input and output stream of a vertex, and adjusts the degree of parallelism of vertices in the graph that has to be scaled in or out. When it comes to scheduling, E-stream monitors whether DAGs require more resources and reschedules them using a priority-based Earliest Finish Time first (EFT) strategy by keeping the system fairness degree guaranteed. E-stream was developed based on Storm.

Floratou et al. [83] focus on service level objectives (SLO) that have to be maintained in case of unpredictable load variation and hardware or software performance degradation. They introduced *Dhalion*, a self-regulating system on top of Heron that collects metrics during runtime to detect possible problems, and apply appropriate policies. The first policy provisions resources dynamically for throughput maximization and the second one, takes as input a throughput SLO, and adjusts the parallelism of application's operators or provides the necessary resources to maintain it.

Safaei [84] presented a whole architecture for a real-time streaming engine in a multiprocessing environment, paying attention to the value of velocity. Queries and some of their characteristics are assigned to the clusters but are accepted if their deadline can be satisfied. System's response time to a query is computed using a function that takes into consideration a number parameters (details about the function can be found in [85]). This performance-aware algorithm compares the execution time of each query to the query's deadline and selects the highest priority query and allocates it to the proper cluster of processors using a First-Fit algorithm. Then each cluster selects from its waiting queue of queries using the EDF (Earliest Deadline First) algorithm. The selected query is then processed in parallel via a proposed deadline-aware dispatching method; Several parameters were checked to evaluate and compare this prototype to Storm while authors mention some penalties as far as it concerns memory usage and tuple losses.

Most of the aforementioned dynamic approaches work with monitoring and re-scheduling which are time-consuming. On the other hand, having overloaded or underutilized machines can lead to increased computations and deteriorate system's performance. While the proposed scheme considers only static scheduling for now, it handles queue waiting times efficiently. Rather than re-configuring online the tasks' allocation to cope with changes in the stream rates as dynamic techniques do, it tries to

maintain a stable and robust configuration by balancing load between the cluster's nodes.

Of course an adaptive version of this scheme would increase its performance, so this extension is left for future work. The main idea behind this extended work is to model the possible system changes (for example, different number of tasks per executor or different number of nodes) as a task redistribution problem, formed by sets of linear Diophantine equations (more details are found in Section 7. Conclusions). Such a strategy would be more comparable to the dynamic schemes that are described above.

3.3 Discussion

Literature review clearly depicts that the existing solutions proposed for determining the applications' scheduling differ in terms of optimization goals, modeling assumptions, and resolution approach. The diversity of the clusters where the evaluation of the aforementioned systems took place does not let us make safe comparisons between them. Moreover, differences in applications' inherent characteristics and in data streams used (e.g. their transfer rate, their realistic nature etc.) are also major impediments to safe comparisons.

The taxonomy provided, reveals that there are more available enhancements that use the operator based model in stream processing than the micro-batch. However, there are enough scheduling approaches for batch processing systems like Spark that might also fit in systems using the micro-batch model but they are not included in this review, since they have not been tested in stream processing. The observed parameters that affect decisions of the presented scheduling techniques are mainly topology, available resources, and workload. System's performance is also important, when it comes to online decisions.

Topological issues: Most scheduling decisions in static approaches rely on the topology that has to be run. When there is not a large processing burden, the system throughput relies heavily on the network communication latencies. Given the topology, communication patterns can be found, inter-node and inter-process traffic can be reduced, and the throughput is then expected to increase drastically. For example, the linear and star topologies (tested in R-Storm) necessarily involve larger number of communications and it is no surprise that the throughput improvements are more significant compared to Storm, when these topologies were tested, since Storm does not take into account the inter-node and inter-process traffic. Assigning the most communicating tasks together on one node or rack (*task-locality*) is a need that differentiates stream processing from batch processing, which pays attention to *data locality* instead (assigning computations to the nodes where the required data is stored).

Resources issues: The processing of big data requires a large amount of CPU cycles, memory, network bandwidth, and disk I/O. Especially in stream processing, memory becomes of crucial importance. It is essential to effectively schedule the tasks, in a manner that minimizes task completion time and increases utilization of resources.

Resource-awareness is a common need both in the static and dynamic approaches studied. Such strategies try to take advantage of node utilization. In GA Storm, the authors have shown that their strategy performs better when the tasks are shared between the maximum amount of cores, thus, we have the fewer possible number of tasks per core. Maximum tasks' performance can be achieved via minimum CPU sharing, consuming though the maximum number of cores. This is an obvious assumption, however it poses a question: How will a resource-aware strategy fully utilize a CPU? As an answer to this question, most strategies presented above (except GA Storm) have to fully assign one CPU per component of the topology. This means that, in an heterogeneous environment, the CPUs with higher computational power should be scheduled to process tasks that require heavier processing, while in an homogeneous environment, scheduling is based on the assumption that the CPUs available have enough capacity.

Cluster heterogeneity affects the static scheduling decisions. Smirnov [69]proved experimentally that throughput is highly determined by the type of CPU. In the aforementioned systems, authors usually indicate if the proposed schemes target at a homogeneous or at a heterogeneous environment. In R-Storm, the nodes on which the tasks are scheduled are determined by a distance function, that is based on the resource availability. However, the scheduling strategy was implemented for homogeneous clusters, where all the CPUs are assumed to have similar computational power. In cases where different CPU architectures exist, resource-aware scheduling cannot be easily applied. For the example of R-Storm, a CPU selected from a group of available CPUs based on the "nearby" available resources criterion, is not guaranteed to work as expected (complete processing), unless it is known in advance that all the CPUs are identical. Aniello et al. [67] have dealt with heterogeneous nodes and considered the CPU speed in their predictions. Their strategy was based on moving tuples across a chain of hops after they have been sent by a spout, until its processing ends up in an ack bolt. As an example of moving tuples, if an executor is taking 10% CPU utilization on a 1GHz CPU and migrates on a node with 2GHz CPU, the CPU utilization would become 5%. With similar CPUs, such a scheduling would require load balancing to increase utilization. However, load balancing necessarily involves careful selection of the tuples assigned to each CPU and some a priori knowledge regarding the size of the tuples. Moreover, Rychly et al. [12] have shown that their resource-aware, worst, standard, and best scheduling approach provide the same results on an homogeneous platform while significant improvement of the best scheduling over the worst and the average scheduling is shown with increasing heterogeneity, because the best schedule fully utilizes the differences in hardware.

Workload issues: When it comes to dynamic approaches, workload seems to influence most scheduling decisions. Workload characteristics become of crucial importance when it comes to micro-batch processing systems as they help to determine the appropriate batch size on scheduling decisions ([65], [64]). Generally, the idea to deal with this issue is to try to adjust the stream sizes accordingly. When smaller streams are used, there is a faster adaptation to system changes. In operator-based systems, workload-aware approaches can help towards mitigating overloading in a worker node. When a

machine's computational resources are not adequate to handle the processing needs, its capacity can be set to a fraction of its actual capacity to prevent overloading (just like T-Storm does) but this is not enough. Large load spikes lead to bottlenecks, possible backpressures and the overall system throughput decreases. Dynamic systems try to face this issue. However, frequent load balancing and state migration techniques can increase overhead and consequently latency. On the other hand, the reduced rate of scheduling can lead to inaccuracy. Workload fluctuations demand elaborate handling.

Elasticity refers to the ability of a cloud to allow a service to allocate additional resources or release resources on demand to match the application's workload. Nevertheless, without adjusting the parallelism of components, a topology's throughput reaches a ceiling above which adding more machines will not improve performance. Scheduling a topology among unnecessary number of machines can cause an increase in communication latency. Elasticity is a matter of crucial importance in online environments as the input rate can vary drastically in streaming applications and operators' replication degree needs to be configured to maintain system's performance. Unfortunately, most of the available solutions require users to manually tune the number of replicas per operator but users usually have limited knowledge about the runtime behavior of the system [6, 14]. Several approaches (e.g. [14, 82, 83, 86–88]) try to deal with replication runtime decisions in stream processing. Elasticity becomes even more challenging in stream processing environments where computations are generally stateful. Guaranteeing fault-tolerance and dynamic load balancing for stateful operators demands state transfer which is quite cumbersome. The interested reader can refer to Hoffmann et al. [89] and Monte et al. [90] for more details on state migration.

Performance issues: Execution environments are not static. Load fluctuations, possible hardware failures and changing network topologies are very common. Overloaded and underutilized machines can deteriorate system's performance. Consequently, systems should be able to handle failures and changes of the execution environment. A system's performance can be either monitored during run-time or derived from old executions' or benchmarks' statistics. Monitoring performance characteristics like total sojourn time (DRS), query execution time (Safaei's approach) or throughput (A-Scheduler), can help systems either adjust their behavior accordingly online or construct models for effective decision making. Dynamic scheduling techniques [14, 18, 63, 67, 76, 80–83, 88, 91, 92] monitor the queue waiting times and performance parameters (e.g. workload, traffic load, system's latency and throughput) during run-time and update tasks' replication degree and their placement [72]). Dynamic techniques, while advantageous, can lead to local optima for individual tasks without regard to global efficiency of the dataflow. This introduces latency and cost overheads or offer weaker guarantees for the desired QoS. The application's reconfiguration and re-balancing, consisting of migrations and scaling operations may also be time-consuming (e.g. $\approx$ 200 secs in Storm [72]), and entail a significant service interruption when it comes to real-time stream processing. Recent works try to develop techniques to deal with application's downtime (e.g. [72,83,87,92]).

As we see, the performance of a DSPS depends on multiple factors. The capacity and the capabilities of the underlying cluster environment in which the processing is

taking place will always limit its performance. Achieving low-latency, high-throughput processing of streams, requires an effective task scheduling that reduces the number of task migrations, allocates the number of dependent and independent tasks in a near optimal manner to decrease the overall computation time of a job, and improves the utilization of cluster resources. It's certainly clear that scheduling constitutes a critical factor for systems' performance.

Prior to providing a scheduling solution, the following chapter presents Apache Storm's preliminaries. Apache Storm's semantics were chosen to describe this work, as it is a mature project, with a very large community and popularity in cloud computing industry, due to its reliability and good processing mode. This approach, though, is generic to any data flow system and suitable for deployment and use in large-scale clusters. As mentioned earlier, Apache Storm is highly scalable with the ability to continue calculations in parallel at the same speed under increased load and reliable. It has been clocked to process 1 million messages of 100 bytes size on a single node which makes it one of the fastest technology platforms.

Chapter 4

Task allocation and scheduling

"Any sufficiently advanced technology is indistinguishable from magic."-Arthur C. Clarke's third law

Apache Storm [17] is an open source, scalable, and fault-tolerant DSPS designed for distributed clusters. As already discussed in previous chapters, its default operator placement policy evenly distributes the processing elements of an application on the available nodes in the cluster, aiming at load sharing but cannot guarantee load balance. It is among the first open source solutions of modern DSPSs, and is used by several research efforts to evaluate scheduling algorithms.

In this chapter, the way Apache Storm represents and executes a stream processing application is initially described in Section 4.1. Its semantics are then going to be used to:

- present in details the mathematical background and problem formulation behind the provided prototype design (Section 4.2)
- provide a task allocation and scheduling scheme as an extension of Storm, that reduces the required buffer space and the inter-node communication costs to increase system's performance, is balanced and periodic, and has linear complexity (Section 4.3).

The application of the derived scheme is depicted using motivating examples in Section 4.4.

4.1 Preliminaries in Storm

Apache Storm allows application developers to write applications that process streams comprised of tuples of data. It represents a streaming application as a Directed Acyclic Graph (DAG), where the vertices show the operators that encapsulate processing logic (called *components* in Storm) and the edges show the data flow direction. Apache Storm uses the terms *topology* for a DAG while a *task* is a component's instance . The

components of a DAG are divided to *spouts* and *bolts*. A spout is a source of streams in a topology and a bolt receives streams, processes them, and forwards them for further processing.

We distinguish between the logical and physical abstraction in Storm. Fig. 4.1 shows the intercommunication of tasks within a common topology in Storm (*logical abstraction*). There is one spout, and four bolts, and each component has four tasks. Links between components in the topology indicate how tuples are passed around.

Part of defining a topology includes specifying for each bolt which streams it should receive as input. A stream grouping defines how a stream should be partitioned among bolts' tasks. Shuffle grouping is the most commonly used grouping [93]. It distributes tuples in a uniform randomly across the tasks. An equal number of tuples should be processed by each bolt. It is ideal when the processing load needs to be distributed uniformly across the tasks and when there is no requirement of any data-driven partitioning. It can be useful for doing atomic operations such as a math operation but in case the operation can't be randomly distributed (e.g. in case of word count), it does not fit.

In the *physical layer*, there is a master node (Storm's default scheduler is a part of the *Nimbus* daemon on the master node) and a set of physical machines (worker nodes) that can host multiple operators. Nimbus communicates and cooperates with a Zookeeper [43] service to maintain a consistent list of active worker nodes and to detect failures. ZooKeeper, is a shared in-memory service for managing configuration information and enabling distributed coordination. Slots on each worker node indicate the number of workers (Java Virtual Machines-JVMs) that can run on this node. The number of slots are typical set to the number of cores and execute a part of the topology. Each worker running is launched and monitored to have possible failures handled by a *Supervisor* executing on its worker node. Supervisors run on each node, to start or terminate workers according to the Nimbus assignments. The Nimbus and Supervisors are themselves stateless. But with Zookeeper, some state information is stored so that things can begin where they were left off if a node crashes or a daemon dies unexpectedly.

Each operator's code is executed by *threads* or *executors* and multiple threads of the same component execute the same computation on different parts of the stream, in parallel. The number of parallel instances of an operator (i.e. replicas) determine the operator's replication degree (also known as *parallelization degree*). In Apache Storm, it is set when a topology is submitted by the user (the interested reader can check Appendix A to see how this can be done in a common application). Consequently, each component in a Storm topology has several executors and each executor can have several tasks. By default, Storm will run one task per thread. Each JVM can host a number of threads from different components of the same topology.

Storm's default scheduler distributes the tasks of bolts and spouts uniformly across all the nodes in the cluster in a round robin fashion, but in this way it is not possible to balance load. Tasks from a single bolt or spout will most likely be placed on different physical machines but the main consideration in this strategy is that the communication between tasks is not taken into account. It is a common assumption that nearby tasks would most probably communicate during processing, so high communication latencies

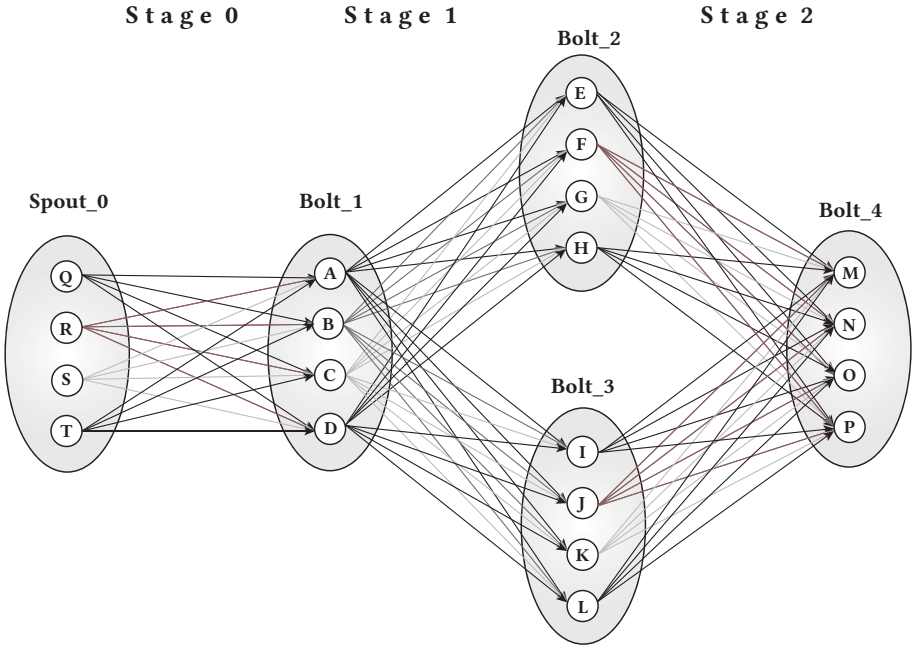


Figure 4.1: Intercommunication of tasks in a streaming application

can be improved, if we achieve task locality.

This approach looks at how components are interconnected within the topology to determine what are the executors (instances) that should be assigned to the same or nearby nodes. The key idea is to use communication patterns among executors, trying to place the most communicating executors, as close as possible. Such a scheduling is executed before the topology is started, so neither the load nor the traffic are taken into account, and consequently no constraint about memory or CPU is considered. Not taking into account these points, obviously limits the effectiveness of an offline scheduler but the pipeline-based scheduling technique that is used, tries to balance load and decrease queue waiting times, enabling a very simple implementation that provides a good performance. It also avoids possible application downtimes, that are usually the case in dynamic approaches to implement the necessary reconfigurations.

Without loss of generality, it is assumed that the tuple processing time of all the tasks is almost the same (this is a logical assumption also used by shuffle grouping, see [13, 72]). Under this hypothesis, we can divide the overall processing into a set of well-defined *processing steps* and *stages*. These terms are defined in the following section.

4.2 Problem Formulation

Let us consider a cluster of N nodes, and an application topology like the one Fig. 4.1, where the interconnection between the components is shown. In this figure, there are

4 bolts and 1 spout, each of them having 4 threads. Each thread executes one task, so we can refer to tasks and threads interchangeably from this moment on. We define the initial *matrix*, M_{init}, as a table that stores the tasks assigned to each node by the default round-robin Storm scheduler. This table can have two forms: in the first form, the tasks are indicated as letters and in the second they have been replaced by numbers.

$$
\begin{aligned}
M_{init} &= \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 \\ \mathtt{Q} & \mathtt{R} & \mathtt{S} & \mathtt{T} & \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} & \mathtt{E} & \mathtt{F} & \mathtt{G} & \mathtt{H} \\ \mathtt{I} & \mathtt{J} & \mathtt{K} & \mathtt{L} & \mathtt{M} & \mathtt{N} \\ \mathtt{O} & \mathtt{P} & \Omega & \Omega & \Omega & \Omega \end{bmatrix} \equiv \\
&\equiv \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & \textcircled{20} & \textcircled{21} & \textcircled{22} & \textcircled{23} \end{bmatrix}
\end{aligned}
\tag{4.1}
$$

The tasks indicated by Ω are added by the model as "dummies" and they are used to avoid empty values in M_{init}. In the numbered representation, the dummy tasks are circled. A dummy task plays no role in the actual processing.

Without loss of generality, we assume that the tuple processing time of all the tasks is almost the same. Under this hypothesis, we can divide the overall processing into a set of well-defined *processing steps* and *stages*. In Definitions 1 and 2, these terms are rigorously defined.

Definition 1: *In our context, a processing step defines a set of communications between nodes (which result in communications between tasks), such that each node receives stream parts from one node only. Once received, these stream parts are assigned to proper threads, which, in their turn, are executed on the data received.*

Definition 2: *In our context, a processing stage defines the points of a logical path of spouts and bolts that a stream has to follow, from its generation until the end of its processing.*

In the example of Fig. 4.1, there are 3 stages: Stage 0, where the stream parts move from the spout to Bolt 1, Stage 1, where stream parts move from Bolt 1 to Bolts 2 and 3, and Stage 2, where stream parts move from Bolts 2 and 3 to Bolt 4. Let us define such an initial task allocation as $\mathcal{A}(t, N)$, where t is the number of tasks per spout/bolt and N is the set of nodes in the cluster. Proposition 1 forms the basis of the task allocation.

Algorithm 1 that is presented in Section 4.3.1 derives directly from the proof of this proposition.

Proposition 1: *The initial matrix of Eq.(4.1) can always be transformed into an intermediate task allocation matrix, M'_{inter}, where the rows of M'_{inter} define a communication between the cluster's nodes, such that each node will be receiving stream parts from one node at a time.*

Proof: If we want to derive a mathematical formula describing the round-robin placement of tasks in M_{init}, this would be

$$M_{init}(i, j) = iN + j, \tag{4.2}$$

where j is the column index, $j \in [0 \dots N-1]$, i is the row index, $i \in \left[0, \dots, \left\lfloor \frac{\mathcal{T}}{N} \right\rfloor - 1\right]$, and $\mathcal{T}$ is the total number of tasks in the system. For our example, the application of Eq.(4.2) would produce the arithmetic representation of Eq.(4.1). By dividing M_{init} by t, we obtain an intermediate matrix, M'_{init}, which represents tasks with their corresponding component IDs (assuming that the spout has ID=0, and bolts 1-4 have IDs 1-4, respectively). The ID=5 corresponds to the dummies:

$$\begin{aligned} M'_{init} &= \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 4 & 4 \\ 4 & 4 & 5 & 5 & 5 & 5 \end{bmatrix} \equiv \\ &\equiv \begin{bmatrix} \mathtt{Q} & \mathtt{R} & \mathtt{S} & \mathtt{T} & \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} & \mathtt{E} & \mathtt{F} & \mathtt{G} & \mathtt{H} \\ \mathtt{I} & \mathtt{J} & \mathtt{K} & \mathtt{L} & \mathtt{M} & \mathtt{N} \\ \mathtt{O} & \mathtt{P} & \Omega & \Omega & \Omega & \Omega \end{bmatrix} \end{aligned} \tag{4.3}$$

Actually, M'_{init} shows the initial allocation of spout/bolt tasks into the system's nodes. Now, $G = gcd(t, N)$ is set as the greatest common divisor of t and N. Thus, we can find integers t' and N', such that $t = t'G$ and $N = N'G$. Under this assumption, let us consider two rows of M'_{init}, one with row index i and one with row index i', where $i' = i + \mu t'$, for some integer μ. Then,

$$M'_{init}(i, j) = \frac{iN + j}{t} \tag{4.4}$$

$$M'_{init}(i', j) = \frac{i'N + j}{t} = \frac{(i + \mu t')N + j}{t} \tag{4.5}$$

By subtracting (4.4) from (4.5), we get

$$\begin{aligned} M'_{init}(i', j) - M'_{init}(i, j) &= \frac{(i + \mu t')N + j - iN - j}{t} \\ &= \frac{iN + \mu t'N + j - iN - j}{t} \\ &= \frac{\mu t'N}{t} = \frac{\mu t'GN'}{t} \end{aligned}$$

All the above divisions are integer. The notion of a *class* will help us proceed with the proof.

Definition 3: *A class k is defined as a group of the row indices of M'_{init}, such that $i \ (mod\ t') = k$.*

Because $\mu t'GN'$ is divided by N', in every column, all the elements of M'_{init} with row indices that differ by t' will produce the same modulo when divided by N'. These elements are G in total and can be brought together by following the steps below:

1. For all the row indices of M'_{init}, find all the k values, such that $i \pmod{t'} = k$. These distinct k values are called *classes*.
2. Take the first row i, such that $i \pmod{t'} = k$, that is i belongs to class k.
3. All the row indices in every class will move to a new row index:

$$i_{new} = \left\lfloor \frac{i}{t'} \right\rfloor + kG \tag{4.6}$$

The row transpositions will generate a *transposed matrix* M'_{trn}. The transposed matrix M'_{trn} has two properties: (1) For every column of M'_{trn}, each class's elements produce the same modulo when divided to N', and (2) M'_{trn} can be divided into a set of $t' \times N'$ sub-matrices of size $G \times G$. In other words, it is a matrix with t' rows and N' columns of square sub-matrices of size $G \times G$. The notation Δ is used to denote these square sub-matirces:

$$M'_{trn} = \begin{bmatrix} \Delta_{0,0} & \Delta_{0,1} & \dots & \Delta_{0,N'-1} \\ \Delta_{1,0} & \Delta_{1,1} & \dots & \Delta_{1,N'-1} \\ \vdots & \vdots & \vdots & \vdots \\ \Delta_{t'-1,0} & \Delta_{t'-1,1} & \dots & \Delta_{t'-1,N'-1} \end{bmatrix} \tag{4.7}$$

each element of M'_{trn} is described by the following indices:

- i row index of M'_{trn} as a whole, $i \in [0, \dots \lfloor \frac{\mathcal{T}}{N} \rfloor - 1]$

- j column index of M'_{trn} as a whole, $j \in [0, \ldots, N-1]$
- r row index of sub-matrices, $r \in [0, \ldots, t'-1]$
- c column index of sub-matrices, $c \in [0, \ldots, N'-1]$
- r' row index of an element within a sub-matrix, $r' \in [0, \ldots, G-1]$
- c' column index of an element within a sub-matrix, $c' \in [0, \ldots, G-1]$

Now, since the elements of M'_{trn} are, in groups, having the same modulo N' values, we can express M'_{trn} as a sum of a constant factor M'_1 and a matrix M'_2, where:

$$M'_1 = \begin{bmatrix} 0 & 0 & \ldots & 0 \\ 1 \times N' & 1 \times N' & \ldots & 1 \times N' \\ \vdots & \vdots & \vdots & \vdots \\ (G-1) \times N' & (G-1) \times N' & \ldots & (G-1) \times N' \end{bmatrix} \tag{4.8}$$

$$M'_2 = \left\lfloor \frac{rN' + c}{t'} \right\rfloor \tag{4.9}$$

Note that M'_1 is a matrix with G in total rows whose elements are multiplied by N', while the M'_2 derives by applying Eq.(4.2) on the elements of a sub-matrix of size $G \times G$, substituting r for i, N' for N, and c for j, and dividing by t' instead of t, $0 \leq r \leq t'-1$ and for $0 \leq c \leq N'-1$.

To complete the proof for Proposition 1, we need to show that the elements of M'_2 can be aligned in such a manner that all the rows contain different elements, since M'_1 is a constant.

Assume that two of the $G \times G$ sub-matrices in the same row of M'_{trn} have the same elements. Because these sub-matrices are a sum of a constant part M'_1 and a variable part M'_2, the variable part M'_2 is examined. Eq.(4.9) states that the row index r of the sub-matrices M'_{trn} has the same modulo when divided by t' but $c \pmod{t'}$ is different for the sub-matrix columns. Therefore, to have a row of different sub-matrices, at most $N'-1$ sub-matrices need to be moved from row r to row r_{new}, using the formula:

$$r_{new} = (rN' + c) (\mathrm{mod}\, t') \tag{4.10}$$

Because $(rN' + c) (\mathrm{mod}\, t') = rN' \ (\mathrm{mod}\, t') + c \ (\mathrm{mod}\, t')$, it follows that the sub-matrices found in the same row have the same $rN' \ (\mathrm{mod}\, t')$ value, but their $c \ (\mathrm{mod}\, t')$ value changes between consecutive columns.

Finally, to separate the same-valued elements found in each row of a sub-matrix because of the constant factor of M'_1 (see Eq.(4.8)), these elements need to be circularly moved to different rows in the sub-matrix, so that they are put in diagonal positions. This can be done because these matrices are square, of size $G \times G$. Thus, every row element will move to a new row index r'_{new} according to:

$$r'_{new} = (r' + c') \ (\mathrm{mod}\, G) \tag{4.11}$$

Now, the elements of every row of M'_{trn} are different between them. The transformations of Eq.(4.10) and Eq.(4.11) produce M'_{inter}. Thus, if we read each row element M'_{inter} as a target node for the corresponding node label in every column (for example, nodes $N_0 - N_5$ in Eq.(4.1)), then each row represents a processing step, where each node communicates with only one other node and Proposition 1 is proven. ■

Proposition 1.1 proves the periodicity of the transformations described and proved above.

Proposition 1.1: *The transformation procedure that leads to a task allocation matrix is periodical and its period is $LCM(t, N)$, the least common multiple of t and N.*

Proof: Assume that two tasks t_λ and t_μ are initially distributed in node n and belong to the same class k. Then, $n = t_\lambda \mod N$ and $n = t_\mu \mod N$. From these two relationships, it follows that

$$(t_\lambda - n) \mod N = 0 \tag{4.12}$$
$$(t_\mu - n) \mod N = 0 \tag{4.13}$$

From Eq.(4.12) and (4.13), it follows that:

$$(t_\lambda - t_\mu) \mod N = 0 \tag{4.14}$$

Suppose that t_λ is located at line i_1 and t_μ is located at line i_2. Since these two tasks belong to the same class k, from the definition of classes, we have: $i_1 \mod t' = k$ and $i_2 \mod t' = k$. From these two relation ships, we have

$$(i_1 - k) \mod t' = 0 \tag{4.15}$$
$$(i_2 - k) \mod t' = 0 \tag{4.16}$$

From Eq.(4.15) and (4.16), we get:

$$\begin{aligned} (i_1 - i_2) \mod t' &= 0 \\ \Rightarrow G(i_1 - i_2) \mod Gt' &= 0 \\ \overset{Gt'=t}{\Rightarrow} G(i_1 - i_2) \mod t &= 0 \end{aligned} \tag{4.17}$$

However, $i_1 = \frac{t_\lambda}{N}$ and $i_2 = \frac{t_\mu}{N}$, so Eq.(4.17) becomes

$$\begin{aligned} \frac{G(t_\lambda - t_\mu)}{N} \mod t &= 0 \\ \overset{N=GN'}{\Rightarrow} \frac{G(t_\lambda - t_\mu)}{GN'} \mod t &= 0 \\ \frac{(t_\lambda - t_\mu)}{N'} \mod t &= 0 \\ \overset{\text{N'=integer}}{\Rightarrow} \frac{N'(t_\lambda - t_\mu)}{N'} \mod t &= 0 \quad or \\ (t_\lambda - t_\mu) \mod t &= 0 \end{aligned} \tag{4.18}$$

Let us rewrite Eq.(4.14) and (4.18):

$$\begin{aligned}(t_\lambda - t_\mu) \quad &\text{mod} \quad N = 0 \\ (t_\lambda - t_\mu) \quad &\text{mod} \quad t = 0,\end{aligned}$$

from which it follows that

$$(t_\lambda - t_\mu) \mod LCM(t, N) = 0 \tag{4.19}$$

Equation 4.19 states that the two tasks differ by the LCM of t and N. Thus all tasks that differ by this quantity can be distributed in the same manner. ■

Proposition 2 forms the basis of task scheduling. Algorithm 2, that is presented in Section 4.3.2, derives directly from the proof of this proposition and along with Algorithm 3 constitute our task scheduling approach.

Proposition 2: *The task allocation defined by the intermediate task allocation matrix, M'_{inter}, can be refined to map to the specific application's DAG, so that the communicating tasks can be (to the maximum extent) placed in nearby nodes and produce the final task allocation-scheduling matrix, M'_{fin}.*

Proof: If we view the elements of rows of M'_{inter} as component IDs (and further, as tasks) rather than target nodes, then Proposition 2 is shown. Indeed, to see that Proposition 2 also holds, one can easily see that each of the Δ square sub-matrices, in its final form, has G elements in each diagonal. In this context a diagonal is a group of G elements that belong to the same component.

By reading the DAG, we can easily see the interconnection between the components. Then, we can interchange the diagonals of the N' sub-matrices over each row of M'_{inter} in a proper way, so that each sub-matrix has component IDs that in fact are settled to communicate by the application. Because M'_{inter} has t' rows and N' columns of square sub-matrices of size $G \times G$, it follows that, every diagonal of each row sub-matrix can be transferred to at most $N' - 1$ different sub-matrices over the row, indication that each diagonal can change position at most $N' - 1$ times. This completes the proof of existence for Proposition 2. ■

4.3 Task Allocation and Scheduling Approach

This thesis' scheme is divided into three parts: task allocation, communication refinement and task scheduling, which are described in the following paragraphs.

4.3.1 Task Allocation

The task allocation strategy is a straight forward implementation of the proof presented for Propositions 1 and 2. In Algorithm 1, the proof of Proposition 1 is organized in

steps, so that the initial task allocation is produced. Then, the diagonal movements discussed to prove the existence of Proposition 2 result in Algorithm 2, that constitutes the *refinement* of the task allocation. The communication refinement of the M'_{inter}, the matrix that derives from the application of Algorithm 1, uses as an input:

- a Bitmap matrix that represents the actual communications between the DAG's components (rows and columns refer to component IDs and "1" is used every time a communication between the corresponding components exists)
- the matrix M' which contains the discrete elements of each $G \times G$ submatrix in each row in ascending order and
- an R^- array that contains the non-communicating components in each $G \times G$ submatrix (each row represents a component ID and its elements depict the component IDs with which it does not communicate, in descending order).

The function $check\_swap$ is called to make the necessary diagonals' interchanges, and gives as an output the refined allocation table M'_{fin}.

Algorithm 1: Task allocation

input : An application graph organized in n spouts/bolts of t tasks A cluster of N nodes

output : An M'_{inter} depicting a task allocation policy, such that each node receives stream parts from one node at a time

begin
Declare M'_{init}, such that $\forall i \in \left[0, \ldots \left\lfloor \frac{\mathcal{T}}{N} \right\rfloor - 1\right]$ and $\forall j \in [0, \ldots, N-1] : M'_{init} = \frac{iN+j}{t}$
Set $G = \gcd(t, N)$, $t' = \frac{t}{G}$, $N' = \frac{N}{G}$
Find all the classes k, such that $i \pmod{t'} = k$
if *a row index* $i \in k$ **then**
  $i_{new} = \frac{i}{t'} + kG$
end
Define the $t' \times N'$ sub-matrices of size $G \times G$
Move all the sub-matrices with similar values to different rows using Eq.(4.10)
Move all the similar values within each sub-matrix to different rows using Eq.(4.11) // M'_{inter} *has been generated*

4.3.2 Task Scheduling

To perform the task scheduling, the inter-node communications need to be arranged. In this context, we can see M'_{fin} as a communication schedule, where each row corresponds to a processing step, as defined in Definition 1. An all to all communication between

Algorithm 2: Communication refinement

input : The task allocation matrix M'_{inter} derived from the application of Algorithm 1 on an application graph
$Bitmap$: A matrix representing the existing communications between components
M': An array consisting the elements of each $G \times G$ submatrix in each one of its rows in ascending order
R^-: An array containing the non-communicating components in each $G \times G$ submatrix in descending order

output : A refined allocation-scheduling table M'_{fin}

```
1  Function main()
2  num_of_swaps = 0
3  for i = 0 to N' − 2 do
4  |   for j = i + 1 to N' − 1 do
5  |   |   check_swap(i,j)
6  |   end
7  end
8  check_swap(i,j)
9  k = 0 // Index used in R^-
10 for x = 0 to G − 1 do
11 |   for y = 0 to G − 1 do
12 |   |   source= M'[i, x]
13 |   |   target= M'[j, y]
14 |   |   if Bitmap[source, target] == 1 then
15 |   |   |   if num_of_swaps < ⌊G/2⌋ then
16 |   |   |   |   swap(R^-[source, k], target)
17 |   |   |   |   k = k + 1
18 |   |   |   |   num_of_swaps = num_of_swaps + 1
19 |   |   |   |   Store to M'_fin
20 |   |   |   |   else
21 |   |   |   |   num_of_swaps = 0
22 |   |   |   |   GoTo Line 3
23 |   |   |   end
24 |   |   end
25 |   end
26 end
```

all the cluster nodes requires N rows for M'_{fin}. In cases where $N > t$, we need to add another $\frac{N-t}{G}$ rows of sub-matrices. Such matrices are always available, since, $N' > t'$ (the number of sub-matrix columns is > than the number of sub-matrix rows). The sub-matrices chosen to be added include the missing communications. This addition

Algorithm 3: Task scheduling

input : Number of stages, s, number of processing steps, P, time required to process a stream part, h

output : M'_{fin} depicting a task scheduling policy with equal processing load per node

```
1  begin
2  δ = 0
3  while processing not complete do
4  {
5     time = δh;
6     for l = 0 to s − 1
7     {
8        if δ − l ≥ 0 then
9           stage_l → step_{δ−l (mod P)}
10          else stage_l → ∅
11    }
12    Execute communications defined by the processing step
13    δ = δ + 1;
14 }
```

results in a square $N \times N$ M'_{fin} matrix and is necessary before applying the scheduling approach of Algorithm 3.

To read the communications, we view the label on top of each column of M'_{fin} as the sending node and the corresponding row elements as the receiving **nodes**. The intra-node communications that result from this scheme are very useful, as they perform task communications internally within a node, thus inter-node communication costs are reduced. Of course we *are also interested that the communication between neighboring nodes is mapped in the columns as well, to the larger possible extent*. This is guaranteed by the way this refinement is performed using Algorithm 2.

Each application runs in processing stages (defined in Definition 2). The task scheduling approach organizes the communication between tasks in a pipeline-based fashion, such that: (1) At each processing step, each node receives stream parts which are delivered to the proper tasks for processing, from one node only, (2) The processing load is balanced between the nodes available.

The time is divided into time slots of duration h, where h is the constant time required to process each stream part. We will use the notation "a step occupies a stage", to show that, from the set of all the stream parts that need to be transferred and processed by the tasks at this stage, the system transfers only the stream parts between the node pairs defined in the step. The rationale between this scheme is the following: At every time slot, different steps occupy different processing stages, thus, each node receives mostly streams processed at different stages of the application (thus, different tasks are occupied). This reduces the buffer space that would be required if a task had to process

many stream parts arriving simultaneously to its node. Algorithm 3 gives the pseudo-code of the task scheduling approach. The application of Algorithms 1,2 and 3 are depicted by the motivating examples of Section 4.4.

4.3.3 Overall Complexity

The transformations required by the provided task allocation and scheduling scheme are of three types:

1. the transformations required to move the classes in the same part of M'_{trn},
2. the transformations of Eq.(4.10-4.11), that will generate the processing steps, and
3. the transformations used for refinement.

The first type is defined by Eq.(4.6) and at most t' rows change position at each class. The transformations described by Eq.(4.10) moves at most $N' - 1$ sub-matrices and the transformations of Eq.(4.11) introduce another G simultaneous moves. Finally, the refinement phase includes at most $N' - 1$ moves of diagonal elements to a new sub-matrix in every row of sub-matrices. Since we have t' such rows, the refinement requires at most $(N' - 1)t'$ moves. Totally, this scheme requires $t' + N' - 1 + (N' - 1)t'$ moves, so its cost is $O(t', N')$, a linear dependence on t' and N'.

4.4 Motivating Examples

We consider two different topologies; a random and a linear as motivating examples. Figs. 4.2 and 4.6 show the interconnection between the components that are used.

4.4.1 Random Topology

In Fig. 4.2 the topology consists of 4 bolts and 1 spout and the maximum number of threads t per component, is 6. Each thread executes one task, so we can refer to tasks and threads interchangeably from this moment on. A cluster of N=9 nodes will be used in this case. Additional dummy threads are added to components that have less than t tasks to define an initial *matrix*, M_{init}, as a table that stores the tasks assigned to each node by the default round-robin Storm scheduler. This table can have two forms: in the first form, the tasks are indicated as letters and in the second they have been replaced by numbers.

$$M_{init} = \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ \mathtt{U} & \mathtt{V} & \mathtt{W} & \mathtt{X} & \mathtt{Y} & \mathtt{Z} & \mathtt{A} & \mathtt{B} & \mathtt{C} \\ \mathtt{D} & \mathtt{E} & \textcircled{A'} & \mathtt{F} & \mathtt{G} & \mathtt{H} & \mathtt{I} & \textcircled{F'} & \textcircled{G'} \\ \mathtt{J} & \mathtt{K} & \mathtt{L} & \mathtt{M} & \textcircled{I'} & \textcircled{K'} & \mathtt{N} & \mathtt{O} & \mathtt{P} \\ \mathtt{Q} & \mathtt{R} & \textcircled{N'} & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \end{bmatrix} \equiv$$

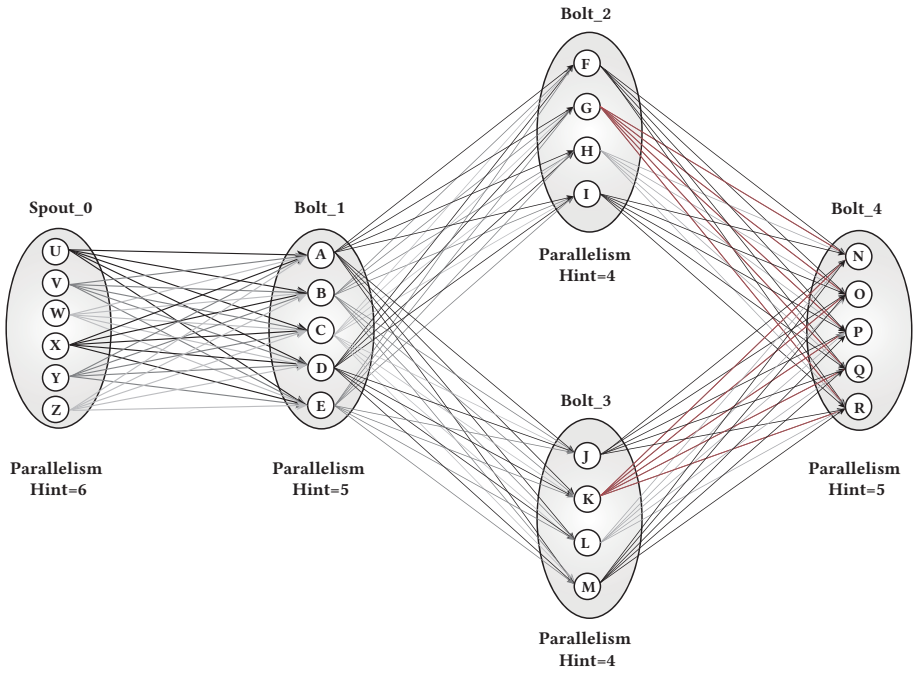


Figure 4.2: Intercommunication of tasks within a random topology (DAG) with heavy communications.

$$\equiv \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & \textcircled{11} & 12 & 13 & 14 & 15 & \textcircled{16} & \textcircled{17} \\ 18 & 19 & 20 & 21 & \textcircled{22} & \textcircled{23} & 24 & 25 & 26 \\ 27 & 28 & \textcircled{29} & \textcircled{30} & \textcircled{31} & \textcircled{32} & \textcircled{33} & \textcircled{34} & \textcircled{35} \\ \textcircled{36} & \textcircled{37} & \textcircled{38} & \textcircled{39} & \textcircled{40} & \textcircled{41} & \textcircled{42} & \textcircled{43} & \textcircled{44} \\ \textcircled{45} & \textcircled{46} & \textcircled{47} & \textcircled{48} & \textcircled{49} & \textcircled{50} & \textcircled{51} & \textcircled{52} & \textcircled{53} \end{bmatrix}$$

The tasks indicated by Ω are added by the model as "dummies", they are used to avoid empty values in M_{init}, and assure that all the desired nodes take part in the allocation procedure. In the numbered representation, the dummy tasks are circled. A dummy task plays no role in the actual processing. The resulting table is an $t \times N$ matrix. By dividing M_{init} by t, we obtain an intermediate matrix, M'_{init}, which represents tasks with their corresponding component IDs (assuming that the spout has ID=0, and bolts 1-4 have IDs 1-4, respectively). The IDs 5, 6, 7 and 8 correspond to components containing dummies:

$$M'_{init} = \left[\begin{array}{cccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 \\ \hline 4 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 7 \\ 7 & 7 & 7 & 8 & 8 & 8 & 8 & 8 & 8 \end{array}\right]$$

We also have $G = gcd(6, 9) = 3$, thus $t' = 2$ and $N' = 3$. Because $t' = 2$, there are two classes, $k = 0$ and $k = 1$. According to Definition 3, row indices 0, 2 and 4 belong to class $k = 0$ and row indices 1, 3 and 5 belong to class $k = 1$. According to Eq.(4.6), row $i = 0$ is the first row of class 0 and as $\lfloor \frac{0}{2} \rfloor + 0 \times 3 = 0$, it will remain in the same position. The next row in class $k = 0$ is the row with $i = 2$. It will move to row $\lfloor \frac{2}{2} \rfloor + 0 \times 3 = 1$. The last row in class $k = 0$ is the row with $i = 4$. It will move to row $\lfloor \frac{4}{2} \rfloor + 0 \times 3 = 2$. For class $k = 1$, we have row $i = 1$, that will move to row $\lfloor \frac{1}{2} \rfloor + 1 \times 3 = 3$, row $i = 3$, that will move to row $\lfloor \frac{3}{2} \rfloor + 1 \times 3 = 4$ and row $i = 5$, that will stay in row $\lfloor \frac{5}{2} \rfloor + 1 \times 3 = 5$. To summarize, we have the following moves:

- Row 0 will stay in row 0
- Row 1 will move to row 3
- Row 2 will move to row 1
- Row 3 will move to row 4
- Row 4 will move to row 2
- Row 5 will stay in row 5

resulting into a transformed matrix M'_{trn}:

$$M'_{trn} = \left[\begin{array}{cccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 7 \\ \hline 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 8 & 8 & 8 & 8 & 8 & 8 \end{array}\right]$$

The horizontal line separates the elements of the two classes. Class $k = 0$ elements are at the top rows and class $k = 1$ elements are at the bottom rows. Also, note that, in every column of M'_{trn}, each classe's elements produce the same modulo when divided to $N' = 3$. Next, we can note that M'_{trn} can be divided into a set of $t' \times N'$ sub-matrices of size $G \times G$:

$$M'_{trn} = \left[\begin{array}{c|c|c} \Delta_{0,0} & \Delta_{0,1} & \Delta_{0,2} \\ \hline \Delta_{1,0} & \Delta_{1,1} & \Delta_{1,2} \end{array}\right] =$$

$$= \left[\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 7 \\ \hline 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 8 & 8 & 8 & 8 & 8 & 8 \end{array}\right]$$

Now, by applying the row transformations of Eq.(4.10), M'_{trn} becomes

$$M'_{trn} = \left[\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 2 & 2 & 2 & 1 & 1 & 1 \\ 3 & 3 & 3 & 5 & 5 & 5 & 4 & 4 & 4 \\ 6 & 6 & 6 & 8 & 8 & 8 & 7 & 7 & 7 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 2 & 2 & 2 \\ 4 & 4 & 4 & 3 & 3 & 3 & 5 & 5 & 5 \\ 7 & 7 & 7 & 6 & 6 & 6 & 8 & 8 & 8 \end{array}\right]$$

and by applying the row transformations of Eq.(4.11), we get the M'_{inter}:

$$M'_{inter} = \begin{array}{ccccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ \hline 0 & 6 & 3 & 2 & 8 & 5 & 1 & 7 & 4 \\ 3 & 0 & 6 & 5 & 2 & 8 & 4 & 1 & 7 \\ 6 & 3 & 0 & 8 & 5 & 2 & 7 & 4 & 1 \\ 1 & 7 & 4 & 0 & 6 & 3 & 2 & 8 & 5 \\ 4 & 1 & 7 & 3 & 0 & 6 & 5 & 2 & 8 \\ 7 & 4 & 1 & 6 & 3 & 0 & 8 & 5 & 2 \end{array}$$

To start the communication refinement, the matrices $Bitmap$, M', and R^- are defined as they are necessary for the implementation of Algorithm 2.

$$Bitmap = \begin{array}{ccccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\text{M'}=\begin{bmatrix} 0 & 3 & 6 \\ 2 & 5 & 8 \\ 1 & 4 & 7 \end{bmatrix},\ \text{R}^{-}=\begin{bmatrix} 6 & 3 \\ 7 & 4 \\ 8 & 5 \\ 6 & 0 \\ 7 & 1 \\ 8 & 2 \\ 3 & 0 \\ 4 & 1 \\ 5 & 2 \end{bmatrix}$$

Based on Algorithm 2, the first swap occurs for $i = 0$, $j = 2$, $x = 0$ and $y = 0$ as:

- $source = M'[0,0] = 0$
- $target = M'[2,0] = 1$
- $Bitmap[0,1] = 1$ as there is a communication from component ID=0 to component ID=1

and this leads to a swap between elements 6 ($R^{-}[0,0]$) and 1 ($target$) in matrix M'_{fin}. Moreover, for $i = 1$, $j = 2$, $x = 0$ and $y = 1$:

- $source = 2$
- $target = 4$
- $Bitmap[2,4] = 1$ as there is a communication from component ID=2 to component ID=4

and this leads to a swap between elements 8 and 4 in matrix $M'fin$. Finally, M'_{fin} becomes:

$$M'_{fin} = \begin{array}{|ccccccccc|} \hline N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ \hline 0 & 1 & 3 & 2 & 4 & 5 & 6 & 7 & 8 \\ 3 & 0 & 1 & 5 & 2 & 4 & 8 & 6 & 7 \\ 1 & 3 & 0 & 4 & 5 & 2 & 7 & 8 & 6 \\ 6 & 7 & 8 & 0 & 1 & 3 & 2 & 4 & 5 \\ 8 & 6 & 7 & 3 & 0 & 1 & 5 & 2 & 4 \\ 7 & 8 & 6 & 1 & 3 & 0 & 4 & 5 & 2 \\ \hline \end{array} \equiv$$

$$\equiv \begin{array}{|ccccccccc|} \hline N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ \hline \texttt{U} & \texttt{A} & \texttt{J} & \texttt{F} & \texttt{N} & \Omega & \Omega & \Omega & \Omega \\ \texttt{K} & \texttt{V} & \texttt{B} & \Omega & \texttt{G} & \texttt{O} & \Omega & \Omega & \Omega \\ \texttt{C} & \texttt{L} & \texttt{W} & \texttt{P} & \Omega & \texttt{H} & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \texttt{X} & \texttt{D} & \texttt{M} & \texttt{I} & \texttt{Q} & \Omega \\ \Omega & \Omega & \Omega & \Omega & \texttt{Y} & \texttt{E} & \Omega & \Omega & \texttt{R} \\ \Omega & \Omega & \Omega & \Omega & \Omega & \texttt{Z} & \Omega & \Omega & \Omega \\ \hline \end{array}$$

Now we can also view M'_{fin} as a communication schedule. An all to all communication between all the cluster nodes requires N rows for M'_{fin}. In our example, we add another $\frac{9-6}{3} = 1$ row of sub-matrices to M'_{fin}, to make it a square 9×9 matrix. The sub-matrices chosen to be added include the missing communications and our scheduling matrix becomes as follows:

$M'_{fin} =$

	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8
step 0	0	1	3	2	4	5	6	7	8
step 1	3	0	1	5	2	4	8	6	7
step 2	1	3	0	4	5	2	7	8	6
step 3	6	7	8	0	1	3	2	4	5
step 4	8	6	7	3	0	1	5	2	4
step 5	7	8	6	1	3	0	4	5	2
step 6	2	4	5	6	7	8	0	1	3
step 7	5	2	4	8	6	7	3	0	1
step 8	4	5	2	7	8	6	1	3	0

The pipeline that derives from the application of Algorithm 3 in the example of Fig. 4.2 is shown in Fig. 4.3(a). Fig. 4.3(b) shows the task communications performed after implementing Step 0, when the pipeline is already full. In our context, the pipeline is *full once all the steps have gone through all the stages once* (unlike the usual interpretation, which states that all the stages are full). When this is the case, we have a communication pattern between the application's tasks, such that the number of multiple part streams loaded to a task is minimized, thus minimizing the need for buffer space. From Fig. 4.3(b), one can see that the communication from task `J` to `P` can only be implemented after the communication from `D` to `J`. The first implementation of step 1, results in a communication between the JVMs of nodes 4 and 2 transferring a stream part from `D` to `J` (task `D` has already received and can deliver a stream part). Then the implementation of step 0, transfers a stream part between the JVMs of nodes 2 and 3 i.e. from task `J` to task `P`. This is depicted by the dashed arrow connecting tasks `D` and `J` in Fig. 4.3(b) as it refers to Step 1.

Moreover, we see a case where possible buffering is required in Step 0; two tasks from node 5 (`M` and `H`) communicate with task `O` in the same node. In such cases (which sometimes are inevitable when multiple components forward tuples to a single one), we could monitor an increase in queue waiting times and tuple latencies. The burden can become even worse when the application workload is increased.

A dynamic scheme could adopt data parallelism and scale out the number of parallel instances for the operator that is overloaded and becomes a bottleneck and/or increase the number of VMs that run in the cluster [72]. Possible task migrations would be needed to reduce resource utilization imbalances between nodes. Elastic data parallelism during run-time, makes a system adaptive to changes in the execution environment but in systems like Apache Storm, the required reconfiguration and restart of the application also results in significant downtime [14, 18].

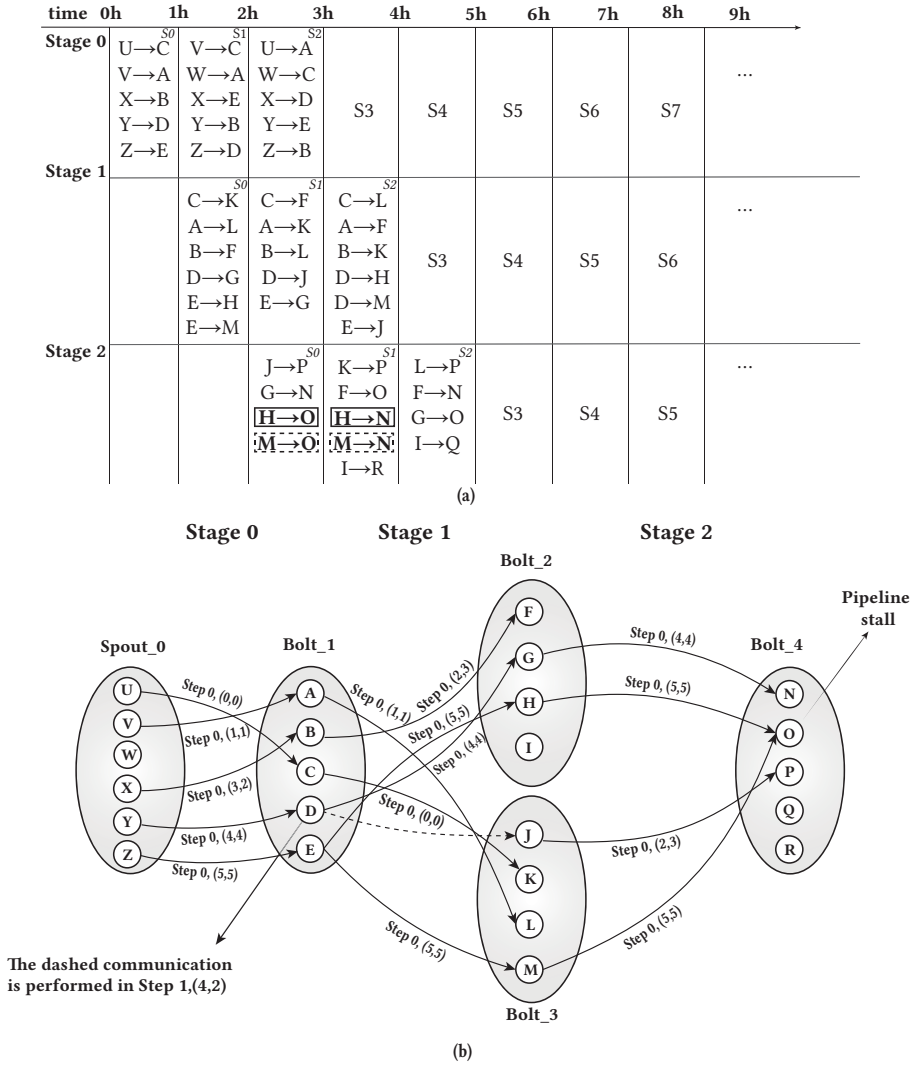


Figure 4.3: (a) Pipeline-based scheduling for the random topology example (b)Task communications after implementing processing step 0, when the pipeline is already full.

In a static scheme, when several tuples arrive simultaneously from different tasks to the same task, more than one tuples could fail to be processed or a tuple could be selected to be processed and let the remaining be processed in the next processing step, where the same communication occurs. In the first case, the tuple could be replayed later or could be missed based on the fault-tolerance guarantees that would be needed in the specific user's topology (at-least-once guarantee in contrast to at-most-once guarantee). This would either increase tuple latency or the system would provide the least desirable outcome in matters of reliability, as messages would be lost [6]. In case of keeping the tuples to be processed in the next appropriate step, the required buffer space would increase.

The presented approach provides an efficient solution for the aforementioned scenario and implements a pipeline stall.The duration of this stall equals to the time required to

process the remaining tuples by the corresponding tasks. In our example the stall's duration equals to h as we just need to let task O process the tuple received from e.g. task M (in case the tuple received by task H was initially selected to be processed).

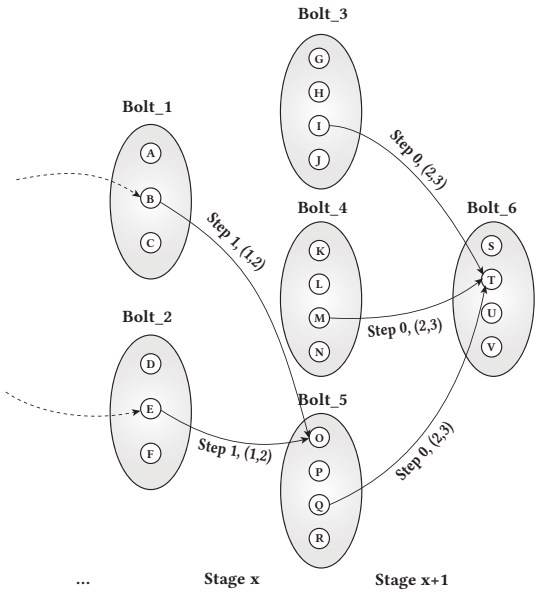


Figure 4.4: A complex scenario

This solution can prove its value in more complex scenarios like the one presented in Fig. 4.4. During Step 0, three different tasks from Node 2 (tasks I, M, and Q) send stream parts to a single task of Node 3 (task T). Simultaneously, in Step 1, two different tasks from Node 1 send stream parts to be processed to a task (O) in Node 2. In this case, the duration of the pipeline stall equals the maximum time needed by a task to process all its stream parts as can be seen in Fig. 4.5. In this case this duration equals $t = 2h$ as O will take $t = h$ to process the remaining part from the second task, while T will need $t = 2h$ to process the remaining parts from the other two tasks that wanted to communicate with it.

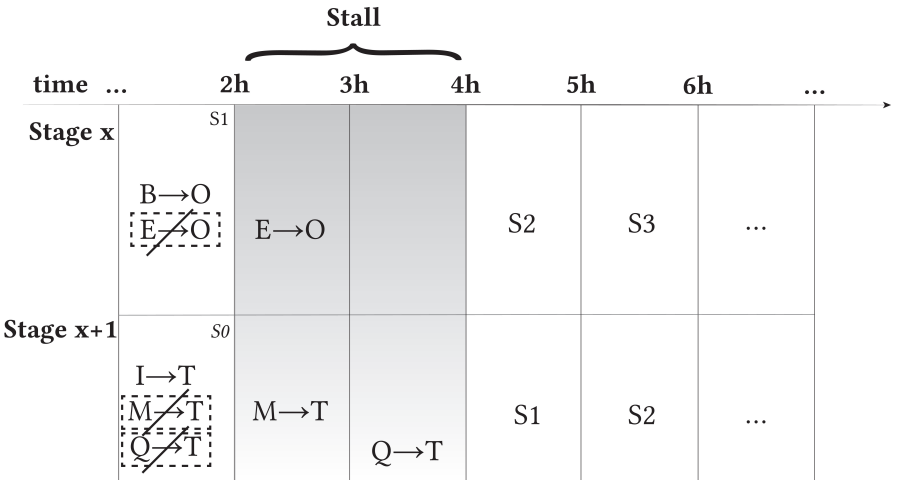


Figure 4.5: Pipeline stall

4.4.2 Linear Topology

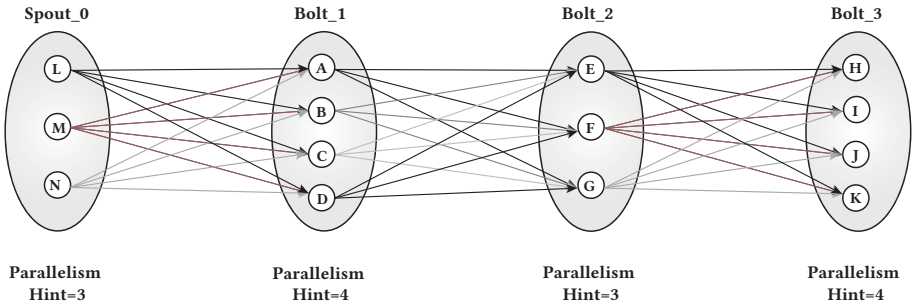


Figure 4.6: Intercommunication of tasks within a linear topology (DAG) with heavy communications.

In Fig. 4.6 a linear topology with 3 bolts and 1 spout is represented. The maximum number of threads t per component, is 4. A cluster of N=8 nodes will be used in this example. The resulting initial *matrix*, M_{init}, is the following:

$$M_{init} = \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \mathtt{L} & \mathtt{M} & \mathtt{N} & \textcircled{\mathtt{L}} & \mathtt{A} & \mathtt{B} & \mathtt{C} & \mathtt{D} \\ \mathtt{E} & \mathtt{F} & \mathtt{G} & \textcircled{\mathtt{E}} & \mathtt{H} & \mathtt{I} & \mathtt{J} & \mathtt{K} \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \end{bmatrix}$$

By dividing M_{init} by t, we obtain an intermediate matrix, M'_{init}. The IDs 4, 5, 6 and 7 correspond to components containing dummies:

$$M'_{init} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 7 & 7 & 7 & 7 \end{bmatrix}$$

In this example we have $G = gcd(4, 8) = 4$, thus $t' = 1$ and $N' = 2$. Because $t' = 1$, there is only one class and consequently, there is no need to transpose lines. The transposed matrix M'_{trn} is equivalent to M'_{init} and can be divided into a set of $t' \times N'$ sub-matrices of size $G \times G$:

$$M'_{trn} = \left[\begin{array}{c|c} \Delta_{0,0} & \Delta_{0,1} \end{array} \right] = \left[\begin{array}{cccc|cccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 7 & 7 & 7 & 7 \end{array} \right]$$

Eq.(4.10) does not result in row transformations in M'_{trn} (there is only one row), but

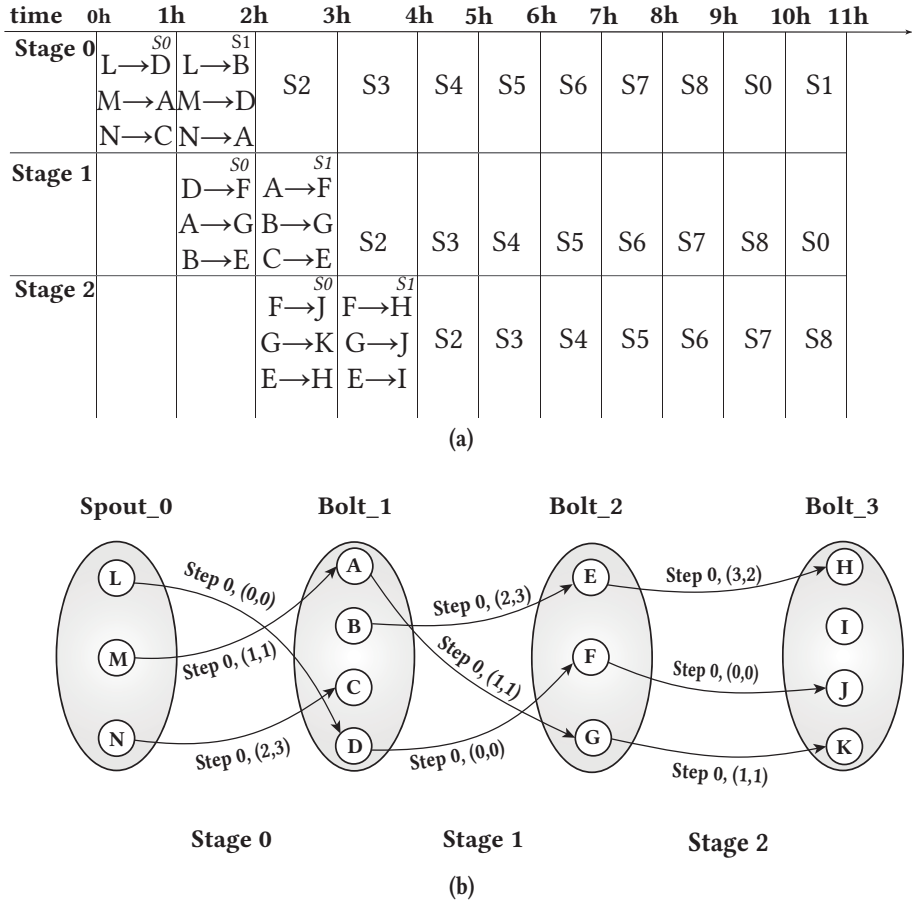


Figure 4.7: (a) Pipeline-based scheduling for the linear topology example (b)Task communications after implementing processing step 0, when the pipeline is already full.

Eq.(4.11), results in the following allocation matrix M'_{inter}:

$$M'_{inter} = \begin{array}{c} \begin{array}{cccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \end{array} \\ \hline \begin{array}{cccccccc} 0 & 6 & 4 & 2 & 1 & 7 & 5 & 3 \\ 2 & 0 & 6 & 4 & 3 & 1 & 7 & 5 \\ 4 & 2 & 0 & 6 & 5 & 3 & 1 & 7 \\ 6 & 4 & 2 & 0 & 7 & 5 & 3 & 1 \end{array} \end{array}$$

The communication refinement algorithm leads to two swaps; 6 with 1, and 4 with 3 that result in the following M'_{fin}:

$$M'_{fin} = \begin{array}{cccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \hline 0 & 1 & 3 & 2 & 6 & 7 & 5 & 4 \\ 2 & 0 & 1 & 3 & 4 & 6 & 7 & 5 \\ 3 & 2 & 0 & 1 & 5 & 4 & 6 & 7 \\ 1 & 3 & 2 & 0 & 7 & 5 & 4 & 6 \end{array} \equiv$$

$$\equiv \begin{array}{cccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \hline L & A & H & E & \Omega & \Omega & \Omega & \Omega \\ F & M & B & I & \Omega & \Omega & \Omega & \Omega \\ J & G & N & C & \Omega & \Omega & \Omega & \Omega \\ D & K & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \end{array}$$

Now we can also view M'_{fin} as a communication schedule that after adding 1 row of sub-matrices to include the missing communications, becomes as follows:

$M'_{fin} =$

	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7
step 0	0	1	3	2	6	7	5	4
step 1	2	0	1	3	4	6	7	5
step 2	3	2	0	1	5	4	6	7
step 3	1	3	2	0	7	5	4	6
step 4	6	7	5	4	0	1	3	2
step 5	4	6	7	5	2	0	1	3
step 6	5	4	6	7	3	2	0	1
step 7	7	5	4	6	1	3	2	0

The pipeline that derives from the application of Algorithm 3 in the example of Fig. 4.6 is shown in Fig. 4.7(a). Fig. 4.7(b) shows the task communications performed after implementing Step 0, when the pipeline is already full.

Chapter 5

Experimental Results

"Everything must be taken into account. If the fact will not fit the theory—let the theory go."-Agatha Cristie

In this section, the efficacy and efficiency of the proposed approach is evaluated under different utilization scenarios. It is also discussed how the performance of this scheduling approach is evaluated to investigate its potentialities and critical points.

At first, the experimental setup is described in Section 5.1. In the following sections, two sets of experiments are presented, as will be described in the following paragraph:

1. The first set of experiments is used to investigate the *average latency* (Section 5.2), the *percentage of buffer memory used* (Section 5.3), the *load balancing per node* (Section 5.4), and finally the *throughput* (Section 5.5) to compare the system's performance against the default Apache Storm's scheduler, using a random and a linear topology.

2. In the second set, the proposed system is compared with one more scheduler, Peng's et al. [19] scheduler, named *R-Storm*, to examine their throughput using a diamond and a linear topology (Section 5.5). R-Storm is chosen as it has been included as an alternative scheduler for Apache Storm, as of v1.0.1.. It tries to reduce the inter communication latency between adjacent tasks just like the presented approach, and takes into consideration memory usage. It is a resource-aware strategy that tries to maximize resources utilization (also discussed in Section 3.2.2.)

5.1 Experimental Setup

The proposed scheduling strategy is evaluated using a simulation environment at GRNET's cloud service ~okeanos-knossos, which provides a wide range of choices to develop, debug, and evaluate an experimental system. Our experimental Storm cluster consists of nodes that run Ubuntu 16.04.3 LTS with an Intel Core i7-8559U Processor system and clock speed at 2.7GHz, 6 CPUs, and 16 Gb RAM per node. Further, there is

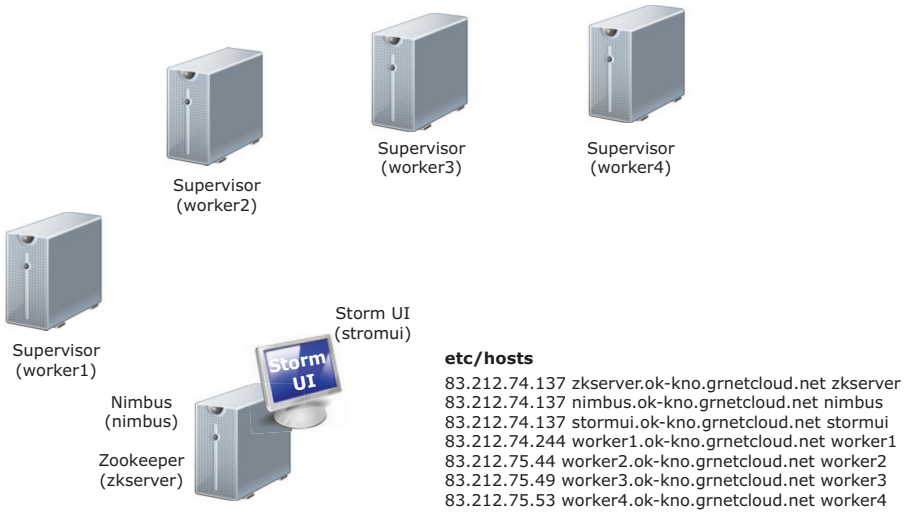


Figure 5.1: Example of Storm Cluster

all-to-all communication between the nodes, which are interconnected at a speed of 100 Mbps. The data transfer rates between the cluster nodes are assumed to be equal, but their proximity differs (nodes with smaller index difference are consider to be located at lower distances between them). The characteristics of the cluster are presented in Table 5.1. The tuples generated are set to have equal size, 8Kb.

Apache Storm 1.2.3 was set up on top of Ubuntu machines along with the prerequisites, JDK 8 and Maven 3.3.9, that is used to build the project. The master node runs three important services Zookeeper (v. 3.4.14), Storm UI, and Nimbus and worker nodes run the Supervisor daemon, as can be seen in the example of Fig.5.1. The corresponding hosts file is also depicted and should be placed in each node to make sure that all nodes can communicate with each other.

Table 5.1: Experimental Environment

Hardware	**CPU**	Intel Core i7-8559U 2.7GHz
	Memory	16Gb
	Network Speed	100 Mbps
Software	**Operating System**	Ubuntu 16.04.3 LTS

The application topology used strongly influences the overall behavior of a scheduler. To conduct the necessary experiments, two topologies were run: (a) A random topology with four bolts and one spout, where the maximum number of threads per component, is 6 (see Fig.4.2) (b) A linear topology with three bolts and one spout. The maximum number of threads t per component, is 4 (see Fig.4.6). For the sake of readability, an example topology written for Apache Storm is postponed to Appendix.

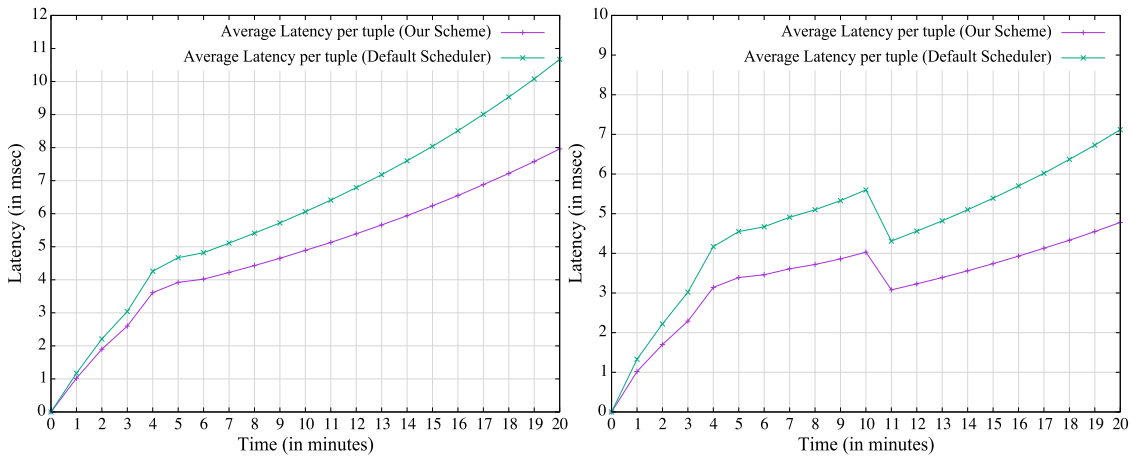
A cluster with $N = 9$ worker nodes, each with 4 slots was used to run both topologies. One extra node, designated as the master node to host the Nimbus and Zookeeper services was also used in both cases. The default Storm scheduler, which is the most widely used

comparison candidate in the literature, was used for the comparisons.

Since 0.8.0 release, Apache Storm allows users to plug in their custom scheduler in order to apply a custom scheduling policy. A custom scheduler has to implement the `IScheduler` interface, which contains two methods; `prepare (Map conf)` and `schedule(Topologies topologies, Cluster cluster)`. The custom scheduling policy is implemented in the `schedule (Topologies topologies, Cluster cluster)` method. Necessary configurations should be made in the `storm.yaml` file of Nimbus to set the new scheduler. On GitHub *(https://github.com/nicoletnt/PipelineStorm)* there is the code used to run a simulation of the propose scheduler, along with necessary configurations done to the nodes, and the needed `.pom` files that contain configuration details, used by Maven to build the project.

5.2 Average Total Latency

This set of experiments focus on the comparison of the overall runtime behavior of our scheduling strategy against the default Storm scheduler. The average latency refers to the time needed by tuples to traverse the entire topology.



(a) (b)

Figure 5.2: Average latency comparisons between our strategy and the default scheduler: (a) Random topology (b) Linear topology.

To fairly estimate the overall latency we worked as follows:

a) Each tuple has to "travel" some distance (nodes with close ID numbers are considered to be placed more closely) from the node that has processed it until the node that will continue the processing.

b) A tuple can either be buffered or directly be processed. However, in cases where there is no buffer space available, the tuple is not omitted. Instead, it is resent after a short period. Although this is not always the case (some systems prefer to omit such tuples), this type of policy can be helpful in examining the overall latency.

In the experiments presented, a buffer space of 32 Mb was assigned per task, which is enough to accommodate about 4K tuples. The other settings (number of threads and nodes) are as described in the previous subsection. Fig.5.2(a) shows the average tuple latency for the random topology, while Fig.5.2(b) shows the average tuple latency for the linear topology. In both cases, our strategy presents lower latencies, but the average gain is higher when the linear topology is used (25% and 40% respectively). There is a combination of reasons behind this result: First, the linear topology has even smaller inter-node traffic when executed in a pipeline fashion, compared to the random topology. However, reducing the inter-node communication cost is not always sufficient to guarantee lower latencies. It is also important to consider that, the proposed strategy avoids having intensively loaded paths between nodes, and this is especially true in the linear case. In the random topology, there are cases where a task may receive large loads (see the example of Fig.4.3). Therefore, this strategy pays-off specifically for linear applications, in that it reduces the overall latency to almost 40% compared to the default scheduler.

5.3 Percentage of Buffer Memory Used

Although there are 3 main resources involved in the overall evaluation of a stream scheduling strategy (network links, CPU and memory), in this paragraph, it is shown that the proposed strategy requires much less memory space compared to the default scheduler.

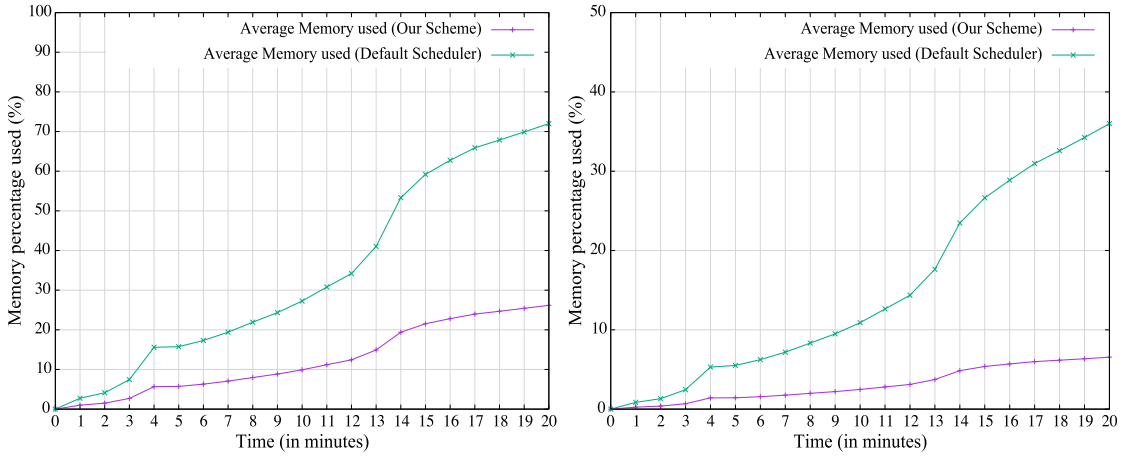


Figure 5.3: Average memory use comparisons between our strategy and the default scheduler: (a) Random topology (b) Linear topology

In the plots of Fig.5.3(a) and 5.3(b), we see the average memory usage for the random and linear topology, respectively. Specifically, when the random topology is executed, the results have indicated that the default scheduler uses, in the worst case, about 70%

of the available memory space available. This is due to the fact that some nodes become overloaded for some periods of time and require more buffering. The proposed strategy uses at most 26% of the memory space, in cases where pipeline stalls may occur (thus buffering is required). For the linear case, the memory space required is reduced for both strategies, however, the improvement offered by the strategy presented, is higher, primarily because only about 7% of the memory resources is consumed in this case (some tuples needed to be bufferred due to network flaws, so re-transmissions were necessary). In fact, in a linear topology, our strategy can have each task receive a tuple at a time thus buffering is not generally required.

5.4 Load Balancing

In this part of the study it is examined whether the developed strategy offers indeed satisfactory balancing between the nodes. The tuple size was reduced to 1KB, to have faster processing time per tuple and the tuples processed at each node were measured.

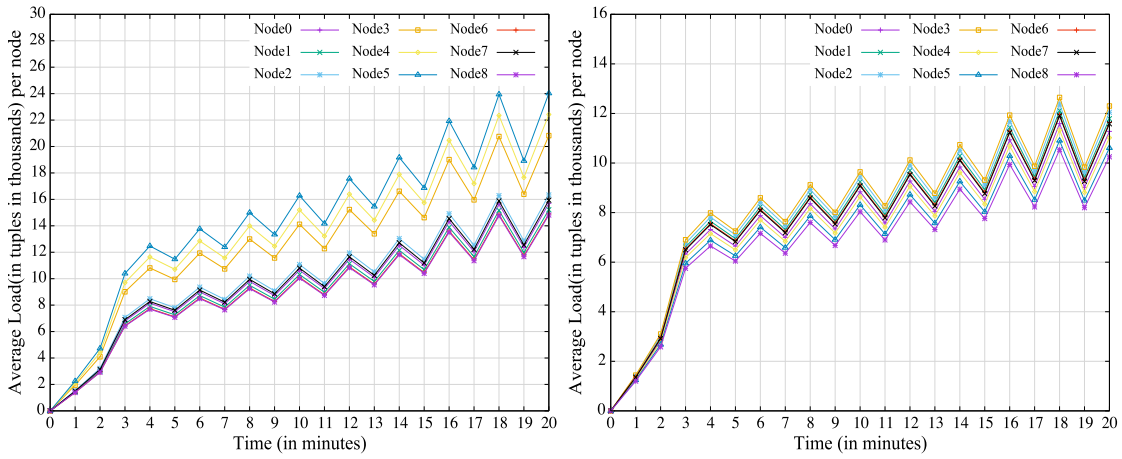


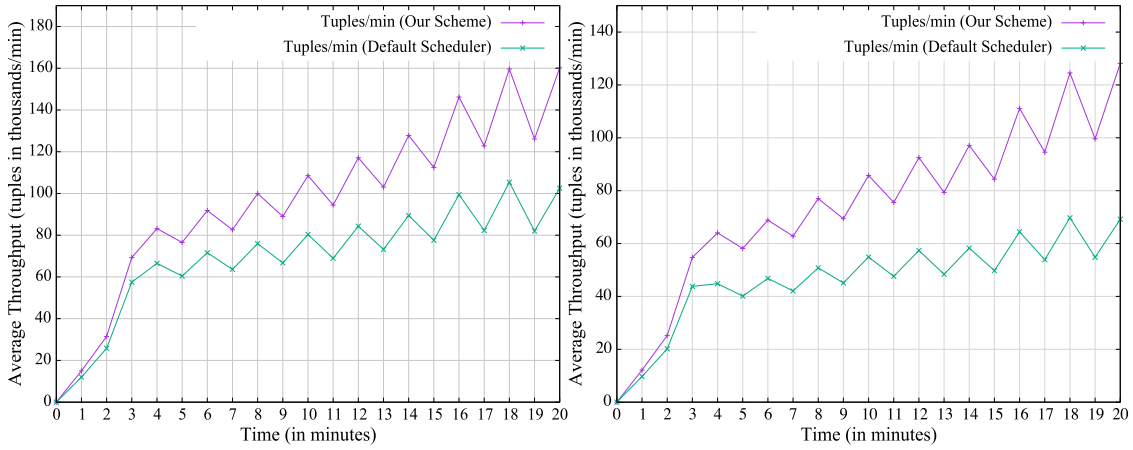
Figure 5.4: Load balancing in a: (a) Random topology (b) Linear topology

The results have shown that, for both topologies, the 9 nodes receive almost balanced processing load (see Fig.5.4(a) and 5.4(b)). Some divergences (an average of 7%) appear in the random topology scenario, where nodes 3 to 5 appear to process more load compared to the others. This can be explained in two ways: (1) these nodes have processed more pipeline stalls (cases where tasks inside these nodes receive more tuples, which are buffered and pipeline stalls are used), (2) these nodes suffer less data losses during transmissions. When the topology is linear, the load delivered to the nodes is more balanced, as seen in Fig.5.4(b): there is a "one-to-one" component communication and "one-to-one" inter-node communication and the small imbalances that appear can be explained by the fact that not all the task communications defined by the proposed scheduler are actually defined in the application. The default Storm scheduler does not

provide any mechanism for handling the communication between tasks in a stepwise manner, so generally it achieves no balancing.

5.5 Throughput

This section provides the results of two different sets of experiments: (i) First, the average throughput (tuples/min) of the designed strategy is compared to the throughput of the default round-robin scheduling strategy. (ii) Then the proposed system is compared to the default scheduler and R-Storm under different scenarios, to verify the fact that it can achieve better throughput performance.
(i) The average throughput is defined as the rate of tuples being processed by the topology's bolts. For the random topology, the number of threads was varied from 3 to 6 and the throughput values were averaged. Tuples were being processed over a period of 20 minutes. Throughput is mainly affected by the inter-node communication required, and the possible delay, when a tuple is buffered to be processed at a later time.



(a) (b)

Figure 5.5: Throughput comparisons between our strategy and the default scheduler: (a) Random topology (b) Linear topology

The results shown in Fig.5.5(a) indicate that the proposed strategy outperforms the default round robin strategy. It offers an average of 25% improvement in throughput. There are two main reasons for this improvement: (i) As the total time increases, the round-robin strategy suffers large numbers of tuples, which are not processed in-time, due to node over-utilization (for example, when multiple tuples are submitted from different tasks to a certain task). These tuples are either buffered, or omitted (in such a case they are re-submitted for processing), and (ii) This strategy places the relevant tasks to the same or nearby nodes (refinement process). This type of placement decreases the total tuple processing time, as the inter-node communication cost is reduced. For larger number of threads and a maximum runtime of 20 minutes, the overall improvement approached 35%.

For the linear topology, the number of threads was varied from 2 to 4 and the results were averaged. Again, the presented strategy outperforms the default scheduler by an average of 40%. Larger throughput increases were noticed compared to the random topology. This is explained by the fact that the developed approach buffers fewer tuples in the linear topology case, compared to the random one. Fig.5.5(b) shows the experimental results for the linear topology case.

(ii) In the second set of experiments (regarding the throughput of our strategy), the proposed strategy was compared with the default Storm scheduler and R-Storm. 8 worker nodes and 1 node designated as the master node, running Nimbus and Zookeeper, were used and two different topologies; a diamond (Fig.5.6(a)) and a linear (Fig.5.6(b)) were run. The diamond topology, consists of one spout, two intermediate bolts one sink bolt. Each component consists of 10 tasks. The linear topology has one spout and four bolts and the number of tasks is 16, 16, 8, 4 and 1 respectively.

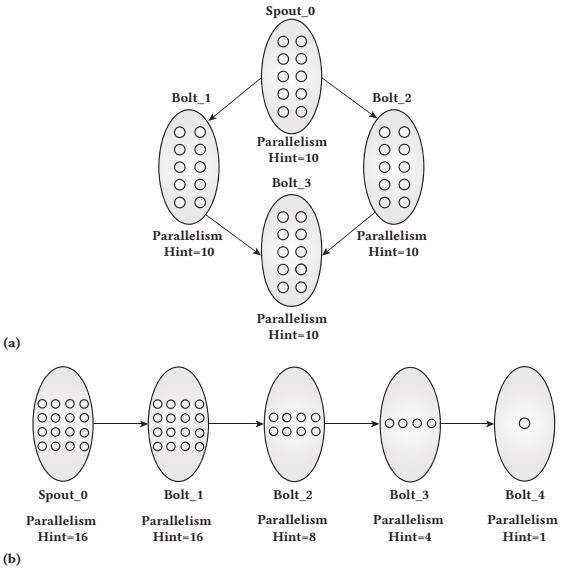
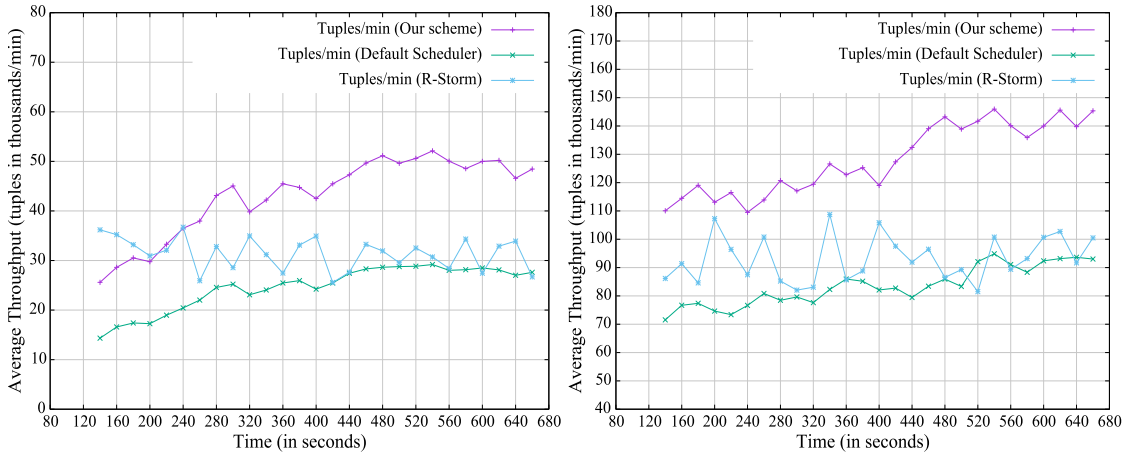


Figure 5.6: Experimental topologies.

The results obtained justify the claim, that when there is a special care on the buffering of incoming tuples, that is, buffering is reduced and thus the highest percentage of tuples are processed as they arrive to the proper target node, then the average throughput increases. The R-Storm does not have any special mechanism for reducing the buffer space required. Instead, it expects the user to provide the node capacities and the task requirements, in order to perform allocation of threads and manage the available resources. From Fig.5.7(a) it can be seen that the provided strategy offers an improvement of $\approx$ 35% compared to R-Storm as the time increases for the diamond topology.

For the linear topology (Fig.5.7(b)) the results indicate that the improvement approaches almost 45% compared to R-Storm. This is because that the developed strategy generally buffers less tuples when running a purely linear topology compared to diamond or random topologies. In such topologies, tasks may receive multiple tuples from other tasks, which can't be processed simultaneously, thus they are bufferred. Then, all bufferred tuples are processed in an extra step (pipeline stall), as explained in the text.

A final observation is that, there is a period of time (in the first seconds of execution) that R-Storm outperforms the proposed strategy. This is explained by the fact that this strategy needs to execute once all the communicating steps and have the pipelines full, in order to start performing more efficiently. When this occurs, the developed scheme clearly outperforms R-Storm.



(a) (b)

Figure 5.7: Throughput comparisons between our strategy, R-Storm and the default scheduler: (a) Diamond topology (b) Linear topology

Chapter 6

An Application Example-Decision Making in IoT-enabled Agriculture

"The goal is to turn data into information, and information into insight."- Carly Fiorina

By 2050, the world's population is expected to be 34% larger than today. According to the Food and Agriculture Organization of the United Nations [94], to keep up with rising population, global food production must increase by 70% in order to feed the world. This poses the challenge of improving agricultural productivity, while lowering its environmental footprint.

Advancements in crop growth modelling, progress in the use of tools to monitor and collect information from farms in a less labor-intensive manner, and global navigation systems give rise to precision agriculture, in which precise measurements at local points and data-intensive approaches support decision-making [95]. Over the past decade, machine learning techniques have been deployed across precision agriculture to provide more accurate solutions, mainly because of the capability to handle highly complex and non-linear agricultural problems [96, 97]. Machine learning techniques have different potential, are of different complexity and computational requirements, and continually evolve. As model complexity increases, more data must be collected.

With the wide application of IoT technology and the amounts of data produced, smart agricultural design can efficiently realize the function of real-time data processing and improve the development of precision agriculture [98].

The Problem: In the process of analyzing and processing a large amount of planting and environmental data, how to extract valuable information from these massive agricultural data, is a pressing problem to be solved.

Machine learning and data mining techniques are expected to be instrumental in meeting the challenges facing global agriculture, and in identifying significant opportunities by taking advantage of big data. However, the collection and analysis of large, complex, heterogeneous data, coming from the variety of sources encountered

in agriculture, cannot be accomplished with traditional machine learning methods such as linear regression. While agronomic models will play a role in the interpretation of data, big data transforms agriculture from model- to data-driven.

To address such knowledge gaps, this chapter:

1. examines how the most promising field, machine learning, is applied to extract information from agricultural data to support precision agriculture in Section 6.1.1
2. presents the current limitations of machine learning to support decision making in agriculture in Section 6.1.2
3. enhances the awareness for the potential implications of big data analytics in agriculture, presenting existing opportunities and promising areas of applications in Section 6.2.1, and
4. sheds light on the factors that delay big data adoption in agriculture, providing future directions, open issues, and research trends in Section 6.2.2, to speed up adoption.

6.1 Precision Agriculture

In traditional agriculture, crops have been treated under the assumptions of uniform soil, nutrient, moisture, weed, and insect conditions. This has several times led to over-applications or under-applications of pesticides, irrigation, fertilizers, and other treatments [99]. The advent of Global Positioning Systems (GPS) and Global Navigation Satellite Systems (GNSS) enabled the practice of precision agriculture, that can be defined as "the collection of real-time data from farm variables and use of analytics for smart decisions in order to maximize yields, minimize environmental impact and reduce cost" [100].

The factual base of precision agriculture is the spatial and temporal variability of soil and crop factors between and within fields. The goal of collecting geo-referenced data is to generate more accurate descriptions of system aspects to inform decisions. New challenges to the successful implementation of precision agriculture stems from technology advances and the huge increase of data both in number of records and variables. The augmented possibilities for data storage, high-throughput, and fully automated technologies have been rapidly generating large-scale data in agricultural settings.

6.1.1 Decision making in Precision Agriculture

Multiple linear regression and linear mixed models have been used in soil mapping, where the variability of a target soil property is explained by its relationships with other soil and climate factors, with shortcomings like autocorrelation and non-linearity between variables [101]. In agricultural practices, a variety of interrelated factors influence crop

production. In complex situations, where data are not linearly related and several outliers exist, linear regression models were not found useful in understanding yield response and did not provide accurate predictions even within sub–field regions, thought to be homogeneous [102–104].

The high complexity and non-linearity of problems faced in agriculture required methods able to approximate complex mappings by integrating data coming from different sources and exploiting the information contained in the obtained reference samples. These methodologies are represented by machine learning techniques [105, 106]. Artificial neural networks (ANN), support vector machines (SVM), decision trees (DT), and random forests (RF) are machine learning techniques frequently applied for agricultural management purposes and the most commonly used in the literature. The decision on which technique to choose depends on the dataset available and the problem's complexity. Existing studies that search for the best technique for specific agricultural aspects do not always present the same conclusions and cannot provide global solutions. Machine learning techniques have been used in several agricultural applications targeting mainly crops, soil, weeds and diseases, and weather/climate change [107].

Crops. Applications targeting crops are mainly cases of yield estimation or the recognition and estimation of crop features. Accurate and timely forecast of yield is required for marketing, storage, and transportation decisions. Machine learning methods, capable of handling non-linear relationships, can process a large number of inputs. Inputs from different sensing systems like soil (e.g., salt, organic matter) or climate characteristics can be combined to predict yield accurately and provide crop recommendations on-time [108–111]. The estimation of crop features' values is also needed in order to better understand the environmental dynamics at a region of interest [112, 113]. Crop recognition is used for the automatic identification and classification of crop species in a fast and cost-effective manner, avoiding the use of human experts [114–117].

Soil. Applications for soil include the estimation of soil components, temperature, and soil moisture content. The knowledge of spatial variability of soil components helps understanding variabilities in production. Accurate estimations of soil properties are needed to optimize soil management, make nutrient planning, and take land-use decisions [118]. Land management practices in agriculture may also target the prevention of floods and landslides to reduce negative environmental impacts. Carbon storage estimation has gained increasing attention in recent years [101, 119], due to its interaction with the earth's climate system. Maintaining and increasing SOC stocks through improved land use and management practices can help to counteract increasing atmospheric carbon dioxide concentrations.

Moreover, soil moisture monitoring enhances the understanding of water exchange rate at the atmosphere/ground interface and has motivated the development of airborne and satellite microwave sensors [120]. Accurate soil moisture estimations can provide high-resolution maps of water content and in tandem with on-time temperature and weather estimates can contribute to the enhancement of irrigation systems and the

maintenance of the climatological balance [121, 122].

Diseases and weed detection. Uniformly applying pesticides or fertilizers over an area of interest leads to high financial and significant environmental cost. Residues in crops, water contamination, and impacts on ecosystems are just some of the consequences of this practice. Machine learning techniques can combine various parameters and perform complex, non-linear modeling of crop yield dependence on nutrients to have optimal agro-chemicals input targeted in terms of time and place [123].

Plant diseases are often associated with several physiological and visual modifications of their host plants, but their visual monitoring at early stages in the field is time-consuming and expensive. Alternative evaluation methods like hyperspectral imaging and non-imaging sensors have proven to be useful for detection of early-stages of vegetation stress, identifying small differences in vegetation cover abundances, or measuring leaf pigment concentrations. For precision plant protection, disease detection methods must facilitate an automatic classification of the diseases [124, 125]. Apart from diseases, weeds are also a serious threat for producers. They are difficult to be detected between crops, but remote sensing technologies allow us to build accurate classifiers, in order to distinguish weeds from both diseased and healthy crops [126–128]. Tools detecting and removing weeds can then minimize the need for herbicides and human intervention.

Weather and climate change. Weather and climate conditions have a profound influence on the growth and yield of crops, affect fertilizer and irrigation requirements, and incidence of pests and diseases. Extreme weather conditions can damage the whole production and cause serious soil erosion, while crop quality during movement from field to storage or to market is severely affected by unpredictable changes in weather. Climate Smart Agriculture is a term that refers to simultaneously improving farm productivity and incomes, increasing adaptive capacity to climate change effects, and reducing green house gas emissions from farming, with the use of an integrated set of technologies and practices [129]. Most persistent issues in this category of applications evolve around capturing the huge heterogeneity of interdisciplinary data. The development of models combining historical data with real-time data collected from several meteorological stations seems promising in providing accurate and in time forecasts to support producers and mitigate the weather effects [130, 131].

While agriculture is strongly affected by climate and weather conditions, it is also one of the economic sectors that strongly affects climate change itself. The relationship between agriculture and climate change is two-way. Precision agriculture can lower emissions by better targeting inputs to spatial and temporal needs of the fields. Improved soil, water, fertilizer, and pest management can significantly reduce greenhouse gas emissions, while maintaining similar yields and reducing production costs. Advanced machine learning techniques proved valuable to imitate the complex, nonlinear issues in the ecological, climatological, and environmental fields [132].

Tantalaki et al. [107] presents an extensive review of research dedicated to applications of machine learning in agricultural production systems with works

categorized based on the aforementioned applications' target. The interested reader can refer to this work to explore where the field of data analysis in agriculture has been, where it is now, and where it is likely to go. The need to manage data that arrives continuously at volumes and high velocity, rises to the top for reasons described below.

6.1.2 Challenges and Limitations

Data-driven models show promise for automating and significantly improving accuracy, computational efficiency, and cost of various tasks in precision agriculture but some issues are raised [105, 133–136]:

- Spatial variability: Spatial variability is of crucial importance to understand the interaction of important variables that affect crop variability. One serious limitation of using the aforementioned models is that they assume homogeneity; fields are not usually homogeneous, leading to false assumptions in yield simulations. The appropriate size of management zone should be carefully identified by experts when making decisions regarding input usage.

- Temporal variability: Appropriate technique selection for each given dataset may vary a lot from one year to another, since new data is always incorporated. Methodologies for optimal data fusion and for making models able to exploit information at different temporal scales are needed to improve the temporal consistency and accuracy of the estimation process.

- Variable selection: It is difficult to establish a constant set of attributes that guarantee good results all the time for all techniques, while some agricultural datasets may also be difficult to model for any technique due to high complexity of the crop behavior. It might make sense that adding more features to the total set of possible features would increase models' performance, but a large number of irrelevant features simply increases the possibility to overfit. The challenge for next generation models includes not only modelling the known factors affecting crop yield but also incorporating all of the important factors. This requires the collection of large and suitable datasets that describe the production process.

- Datasets availability: Complicated models with many features compared to the training examples, are likely to overfit. The application of machine learning methods combined with sensing technologies, conducted on small areas with small samples of data, leads to a low ability to generalize the learned parameters to areas with different characteristics. The availability of large datasets from diverse sources is necessary to achieve better generalization.

Farming systems are affected by various factors like environmental conditions, soil characteristics, managing of crop diseases and weeds, and water availability. Lack of data restricts the capabilities of existing models to include factors of importance and be accurate enough to gain users' confidence in their abilities to provide reliable

results. However, the amount of data collected on farms through sensors like yield monitors, drones, or portable devices has increased dramatically over the last decade. The availability of high quality spectral, spatial, and temporal resolution data can lead to refined and robust models. The types of data have also changed, because, apart from simple numerical values, data may include qualitative measures, images, or videos. The desire to collect information on soil and crop variability and respond to such variability on a fine-scale has become the goal of precision agriculture. The use of big data aims at supporting this goal and the use of real-time analytics for on-time decision making will further provide the competitive advantage needed for farmers who adopt the IoT ecosystem. However, even when large datasets can be available, machine learning techniques that focus on learning from data, still have to face challenges, as described in Section 6.2.2.

6.2 Big Data in Agriculture

The emergence of trends like the IoT, robotics, and cloud computing allowed for an increase in the volume, velocity and variety of data generated in agriculture. Metadata capturing management practices and technologies, such as seeding depth, seed placement, cultivar, machinery diagnostics, time and motion, dates of tillage, planting, scouting, spraying, and input application are considered big data in agriculture [137]. We consider that big data analysis in agriculture does not need to satisfy all the three V's dimensions–velocity, volume and variety. Based on precision agriculture applications, the use of sensors and GNSS to create spatial variability maps can lead to high volumes of data. The highest volume appears in remote sensing applications because of the large sizes of the images used. Taking into consideration that weed and disease detection require urgent action, relevant projects and alert systems demand high velocity. Nevertheless, soil and crop related approaches for production estimations do not demand immediate actions and rarely have to deal with data of high velocity. Decisions on weather forecasting (e.g., decisions for irrigation or fertilization) also need to be made at almost real time. Papers referring to weeds and diseases, dealing with production security, do not have to access a variety of data to address a problematic issue. On the other hand, modeling of weather and climate change needs various data sources to provide accurate forecasting and support producers' tasks [95]. Innovative technical and analytical strategies have been developed to cope with such data and are gradually gaining popularity.

Both big data and precision agriculture derived from the advent and application of information and communication technologies (ICT), yet they are not synonymous. Precision agriculture involves site-specific application of inputs and the use of yield monitors. It employs graphical comparisons of field maps as its dominant method of analysis. However, identifying complex interactions across several production factors and multiple years requires more sophisticated methods. Analytics is a major differentiating feature of big data and methods for their implementation are presented in Section 6.2.2.2. Despite these differences, precision agriculture provides an input

for big data for analytics. Big data analytical platforms in the cloud, and machine learning techniques that drive artificial intelligence are helpful, when fully realized [95,105,138–140]. We can consider precision agriculture and big data as complementary to each other. In the following section, the agricultural big data systems are grouped into three categories. Each category is described, and application examples and promising areas of big data application are presented, based on the research conducted.

6.2.1 Agricultural Big Data Systems

Agricultural big data systems can be divided into three categories. Most of the applications found in the literature use advanced machine learning techniques. Different approaches are used in several agricultural areas. Specifically, these systems take advantage of IoT technologies but have different scopes. As such, we divide them into three domains:

- **Advanced sensor technology systems** - refers to systems that collect data to characterize spatial and temporal variability in the production system and determine actions to be taken in field.
- **Risk management systems** - refers to systems that use advanced analytic techniques to manage the risk of crop failure. These systems attempt to make risk management specific to field location, soil type, and desired yields and assess the most probable risks on a given farm. Weather and climate change adaptation and mitigation are common matters of interest in such systems.
- **Agricultural management systems** - refers to systems that provide smart farming solutions. They address farm needs like accounting, food market access and traceability, and wireless linking of farm managers, operators, consumers, and stakeholders, to provide support for better management practices.

6.2.1.1 Advanced sensor technology systems

Remote sensing provides efficient ways to collect information over very large geographical areas. The availability of spatially and temporally referenced input and output data, incorporating the effects of climate and soil on yields, allows rapid and accurate estimation of production relationships and surpasses the traditional experimental approach [106, 134]. Production systems, deploying robotics, advanced sensors, and big data analytics, enables farmers to manage their farms on much smaller, and consequently more precise, scales [139–141]. Real-time data processing technologies have seen an unprecedented growth due to their crucial impacts by providing live monitoring and real-time analytics. The resultant analysis can support the automation of numerous agriculture procedures.

Several firms are active with precision agriculture trials using environmental sensors and big data analytics software to maximize yields at a reduced cost [142, 143]. For instance, Monsanto that purchased Climate Corporation for its weather data and modeling

technology, and John Deere that bought Precision Planting to increase the machine learning capabilities of its farm equipment [140, 144]. There are also open source projects that focus in farm automation. For example, Handsfreehectare [145] is a project that aims to use solely automated machines in order to grow arable crops remotely.

Using images from remote sensing instruments like drones or satellites is a recent practice to approximate agriculture problems by image analysis [146, 147]. Generating accurate but also timely maps with high spatial resolution using image samplings remains a scientific challenge in agriculture. For instance, [148] used Landsat multi-temporal scenes and took advantage of the short-wave infrared bands that proved to be extremely useful in efficiently identifying differences between crops. They combined the newly acquired data with spectral data from Landsat satellites over a 15-year period and used supercomputers to handle the huge amount of data. Their Deep Neural Network (DNN) managed to distinguish the crops studied and estimate production with 95% overall accuracy just two to three months after planting. Their approach seems promising enough to be scaled up to large geographic extents.

Object recognition and classification from aerial and satellite imagery using DNN is one of the most promising areas of big data application in agriculture. Crops are systems with increasing complexity in shape and appearance. CNNs have shown excellent capabilities in extracting useful information from images. Moreover, the learned features obtained from pretrained CNN models can generalize properly even in different domains for those in which they were trained [149]. In-time accurate maps derived from high spatial resolution satellites can determine apart from growing conditions, and threats to support best farm management practices on a local scale [148].

So far, methods to collect and process agricultural data from monitoring devices were usually time-consuming as demanded either to get the data manually from the device or use a proprietary cloud as a broker. However, the increasing need to process the growing size of the generated data on-the-fly led to a shift from storing and processing historic data towards using event-driven and scalable web architectures. For instance, Wang et al. [150] proposed an infrastructure to allow the continuous agricultural machine data flow to the cloud to automate farming processes. The processed data successively stream to various external endpoints such as mobile or web applications for interpretations and visualizations. Real-time alerts and feedback for diagnosis, maintenance and monitoring of various agricultural aspects can in this way, be immediately provided.

6.2.1.2 Risk management systems

Management of risk due to field location, soil type, and mainly to heat stress or freeze is a matter of crucial importance in agriculture. A specific circumstance for farming is the influence of the weather and especially its volatility. Merging datasets is a key operation for data analytics in this case. Regional climate models are used to combine information from global models with regional and local meteorological records to provide climate information for smaller spatial units and support real-time adaptation to climate and weather changes [129, 134, 141].

Math et al. [151] proposed an IoT based real-time local weather station to support

precision agriculture activities. Their system provides farmers with means to automate common agricultural practices like irrigation, fertilization and harvesting at the right time. The proposed system also supports the efficient use of agricultural resources at the the right time, when needed by the crops while providing predictions of weather conditions in near future. Towards this direction, Climate Corporation's platforms use in-season imagery and DL to help producers identify issues early and take action to protect and improve yield [152].

A promising domain of big data application in agriculture that relies in this category, although it is not based on real-time processing, is the facilitation of agriculture insurance policies. Weather satellites with wide-area coverage, used in tandem with accurate big data machine learning algorithms that combine the collected meteorological data with auxiliary data (e.g., planting/production records over different areas) can be employed to predict possible crop failures across large regions [129]. In this way, insurers can improve the prediction of potential crop performance beyond what weather alone might allow. When greater insight and understanding of crop production risk is developed, better risk management solutions and personalized insurance policies can be offered, and the risk can be priced accurately. Machine learning techniques are valuable in such systems due to their ability to handle heterogeneous data and capture nonlinear and high-order interactions in dynamic environments [153]. Several attempts have been made recently to establish insurance programs in developing countries [154, 155].

6.2.1.3 Agricultural management systems

ICT enables farmers to exchange information, establish cooperation, and collaborate. As farmers get connected, software management systems emerge. Agricultural management systems arise to provide accounting services, linking farmers with farm managers and operators, and give benchmarking abilities to farmers by connecting them. Their aim is to help farm operators and agribusinesses around the world collect, integrate, and analyze huge amounts of data from different sources to support their business decisions. Such systems provide smart farming solutions. Smart farming is a term that extends precision agriculture by basing management tasks not only on field-specific data, but also on data enhanced by context and situation awareness, triggered by real-time events [139]. For instance, studies conducted in developing world small farms indicate that farmers are not able to sell harvests due to oversupply or lack of necessary information [156]. Tools for better yield and demand predictions can enable crops to be integrated to the international supply chain [95]. Marcu et al. [157] provide a comparative study between the most common agriculture IoT platforms that enable farmers to perform real-time actions and bring together the users (farmers) and professional suppliers.

Singh et al. [158] proposed a big data analytics approach that collects and analyzes social media data using vector machines to identify issues related with supply chain management in food industries. Their approach led to a cluster of words informing supply chain decision makers about ways to improve various segments of food supply and thus support production planning and scheduling. Social media text analytics can inform decision makers about improving various segments of food supply chain

management [95, 158]. Demand and supply are affected by many unpredictable factors that interact in a complex manner. Analyzing and interpreting details on market behavior and consumers' preferences from several sources in real-time can assist producers in making better and faster decisions to satisfy customer requirements. Moreover, spatial mining techniques on collected data can be used to identify regions susceptible to possible disasters (e.g., severe weather conditions), to predict locations inappropriate for sensitive crops, and update the supply chain accordingly [138].

Table 6.1 presents the opportunities provided by the use of big data in agriculture and the potential benefits. Collecting and analyzing big data generated by automated systems, including digital images and other data from ground sensors, unmanned systems, or remote sensing satellites, and their combination with already existing data pose challenges to successful implementation of precision agriculture. Emerging fields of data mining and machine learning methods are promising approaches to gain insight from such data [95, 97, 138, 140, 159]. These methods can help analyzing bigger and more complex data, to uncover hidden patterns and reveal trends fast and accurately. The potential of these techniques in big data analysis, though, have not been adequately appreciated in agriculture for a number of reasons examined below.

6.2.2 Challenges of Big Data Adoption in Agriculture

Most of the available open platforms mentioned previously result from recent projects; their challenge is still the broad adoption, to determine final success. Many of them may still be under development and have not reached their full potential yet. There are several publications describing precision agriculture but reports with evaluation of the economics of big data adoption in agriculture are much less numerous [95, 160]. However, systems' marketplace adoption can be monitored as a means to assess whether there are benefits from their technology.

Several big data applications seem to be suited to large farms and industries (i.e., Climate Corp and Monsanto) that already use data in their decision-making and have access to data captured from machinery, greater access to capital, and resources [152]. Smaller intercropped fields, though, may require more manual labor and less mechanized processes. However, there is little research examining this assumption [161, 162]. Big data could potentially be very useful for non-industrial farming practices, but emerging moral and ethical questions about access, cost, and support should be addressed to realize this benefit. During this initial phase, benefits from data are not so large for the farmers. Concerns are held among growers, that the benefits and risks of big data related developments will be unevenly distributed. Concerted efforts are needed to lay the foundations required for everyone who wants to participate, to be able to participate [161]. This involves at least improving access to Internet connections even in very remote areas, and by not excluding smaller farmers (e.g., due to high start-up costs and complex contract arrangements) [162]. The interested reader in these concerns can check [152, 161, 162].

Expectations from the big data adoption in agriculture are high but this adoption is relatively slow. Next, the challenges that have to be faced to leverage the value that big

data have to offer in agriculture are described.

6.2.2.1 Data collection

In agricultural applications, big data comes from various sources either in real-time or not. Combining data from a variety of sources raises concerns about matters of data quality and data fusion, and the access to collected big data raises concerns about security and privacy.

Data-driven methods demand clean and relevant data to be utilized. Incomplete datasets, destroyed data, and the presence of outliers or biases in the training set affect models' accuracies. The assessment of data quality demands significant human involvement and expert knowledge. Even semi-automated approaches are not practical when it comes to large volumes of data. Event monitoring and real-time processing of streams of data that are highly applied in agriculture further deteriorate data quality. Techniques like outlier detection, data transformations, cross-validation, and bootstrapping are valuable tools in data quality management, but until recently, data quality research has primarily focused on structured data stored in relational databases and file systems.

IoT data in agriculture usually comes in streams from sources in geographical proximity and is more likely to be correlated. The spatiotemporal correlation of data permits advanced sensing techniques like compressive sensing to minimize the sampling rate and consequently the network traffic load [163, 164], but demand real-time anomaly detection algorithms [165]. Research on data quality management is ongoing. Computational techniques to tackle the aforementioned challenges are needed [40].

The traditional multi-source data fusion just handles structured data [166]. However, the availability of large datasets is necessary to help data-driven models achieve better generalization. Recent advancements in cloud computing and distributed processing for voluminous data computing could help integrate resources in different scales but this is insufficient. New methods are also needed to tackle the challenges of data fusion, representation, and cleansing but this still remains an obstacle for the exploitation of IoT big data in agriculture [167]. Deep learning models have been shown to be very effective in integrating data from different sources and can handle successfully representation problems like the "semantic gap" [134, 168, 169].

The practice of big data collection also raises concerns over access and security. The ability of researchers to conduct large scale and big data oriented research strongly depends on the availability of farm data. Ag-Analytics [170] is a platform that supports stakeholders for this purpose. Recently, more big datasets are becoming publicly available [171, 172]. Nevertheless, data sharing demands special attention to matters of data privacy and security [173]. Recommendations for governing security, data ownership, data protection, and data use should be set by farm alliances and agriculture technology providers. Federal legislation protecting farm data is also required.

6.2.2.2 Analysis techniques

Big data needs extraordinary techniques to efficiently process its large volume within limited run times. Hypothesis testing and machine learning are the most commonly used ways for data analysis [174]. Agricultural analysis is largely statistical. Its main intention is to understand the underlying system through an analysis of observations. Such approaches start with a theory and lead to one or more hypotheses. Statistical significance tests try to extract conclusions for the population using small samples of as much as possible high quality data. Nevertheless, in the case of big data, the sample used may represent even the entire population. The underlying concept of big data relies more on correlation and less on causation [106, 175]. However, there are examples where the two are effectively combined [174].

Machine learning techniques do not use preconceived relationships from theory but begin with the data, to examine possible relationships among variables [79, 106, 175]. Nevertheless, the collected datasets are large and complex making it difficult to deal with typical machine learning techniques. Such techniques often perform poorly when applied to agricultural data. Scalable and parallel techniques are needed to cope with voluminous data. Moreover, big data collected in agriculture violate common assumptions underlying several machine learning and analytics methods, such as the independence and identical distribution of data (i.i.d assumptions). Big data in agriculture exhibits spatio-temporal autocorrelation, has heterogeneity and high dimensionality, is nonstationary, and usually has to be processed in a real-time manner [106, 138, 176].

For instance, if we consider adjacent plots, we will find similar soil-type, climate, and precipitation. Models that are inaccurate or inconsistent with the dataset may be extracted, if we ignore auto-correlation during data analysis. A variety of spatiotemporal methods used for traditional data could be extended to handle agricultural big data [177–179].

The unstructured streaming data received from several diverse agricultural sources are multi-dimensional. Having many dimensions gives rise to accumulated error terms and there is no guarantee that every dimension is especially useful for performing analysis [106, 138, 176]. Statistical and machine learning techniques do not lower the dimensionality of the problem in a deterministically exact way and they exhibit the "curse of dimensionality" [178]. The are several techniques in handling high-dimensional data like Principal Component Analysis and Incremental Singular Value Decomposition, but most of them are based on dimension reduction and usually fail to extract the core value from massive big data [79].

Moreover, machine learning models are not appropriate for non-stationarity (i.e., cannot be used when new climate and weather patterns or new and improved crops arise), but big data in agriculture exhibit non-stationarity. Most methodologies learn through historical datasets. Consequently, models trained on specific observations should be combined with mechanistic models based on theory and domain knowledge to explore explanatory relationships [138]. Since many agricultural applications are time-sensitive and depend on data freshness, developed models must be retrained to reflect the evolution of data. For learning from high speed streams of data, online learning should be

integrated with traditional techniques and theory [178].

Several current advanced machine learning models have gained a considerable amount of interest as promising frameworks for handling big data in agriculture. Deep learning is of crucial importance in providing predictive analytics solutions for large-scale datasets, especially with the increased processing power and the advances in graphics processors. DNNs can work with thousands of parameters, but complex models can overfit easily. Increasing the dataset to model the interactions among production variables at different locations and seasons could relieve the overfitting problem but can be unrealistic and costly. RFs using multiple predictor functions (to avoid using just one overfitted function) and kernel methods like SVMs could be useful solutions to avoid overfitting, but SVMs suffer from serious scalability problems in both memory use and computation time [79, 97]. To speed up learning, the novel learning algorithm, extreme learning machine (ELM) is proposed to deal with high velocity of data; it is able to provide extremely fast learning speed and achieve better generalization.

To handle the big datasets that accompany precision agriculture, analytics methods must scale up in parallel and distributed ways, avoiding high computational complexity. Advances in cloud computing and parallel/distributed architectures can help towards this direction. Cloud computing can be used to integrate sources in different locations, and then the data input can be partitioned into a distributed and parallel architecture. The combination of machine learning and parallel training implementation techniques provides potential ways to process big data. Developed models, though, should be compatible with parallel computing; unfortunately, not all algorithms can be distributed or implemented in parallel form. As a successful example, Parallel SVM (PSVM) [180] reduces memory and time consumption. In Baldominos et al. [181] a scalable machine learning service is introduced for stream processing and real-time analysis.

6.2.2.3 Computing infrastructure

Big data demand not only novel analytical paradigms to extract information, but also compatible parallel computing frameworks and novel wireless solutions, that may be too elaborate for an individual farmer. In farm management, several technical challenges exist, as diverse and high-dimensional data streams from sensors should be ingested in real time, delivered and analyzed usually in short time, to meet the demands posed by several agricultural applications [134]. Real-time analysis platforms are needed to deal with online remote sensing data and combine it with offline data from one or more distributed data centers. Precision agriculture relies heavily on event monitoring that demands data stream processing and consequently requires lower latency and higher bandwidth. Also the amount of disk input/output (I/O) has to be minimized.

The parallelism of Hadoop (the most used open source implementation of MapReduce) [4] is suitable for batch processing and products for performing advanced analytics on stored big data have largely been built over Hadoop. Nevertheless, Hadoop is not appropriate for nearly real-time routines due to its disk I/O intensiveness as already discussed in the previous chapters. In-memory computing eliminates significant amount of disk I/Os and thus reduces data processing time, enabling the immediate

analysis of live data. Real-time stream processing engines like Apache Storm [17], Spark Streaming [29] and Flink [30] have been developed for this purpose. Modules that can work with these systems like Mlib [182] and GraphLab [183] provide common machine learning operations, while others like Tensorflow [184] are designed to build sophisticated machine learning models like DNNs in real-time. An initial prototype that combines the aforementioned tools to provide real-time analytics in farm is presented in [185] and has been tested with various farms showing prominent results.

6.2.2.4 Storage and interpretation

Results are delivered to target destinations like databases, micro-services, and messaging systems via supported application programming interfaces. Cloud computing, apart from realizing the needed scalability for machine learning algorithms, can also enhance storage capacity through necessary infrastructure. Big data storage led to the development of NoSQL databases (e.g. Hbase [186], Cassandra [187], and MongoDB [188]). Many big data tools rely on open source software solutions, which dramatically reduces costs, but expenditures related to hardware, its maintenance, and the training of potential users are matters still to be faced.

Massive amounts of data cannot be interpreted by producers. Big data and its resultant analysis will not have much impact unless it is understood, adopted, and adapted by farmers and other managers. Visualization is a key component of services intended to enhance precision agriculture. Techniques are needed to make analytics work for producers and help them act on events as they happen. Visualization should be considered as early as possible and in tandem with prior interdisciplinary domain expert knowledge, provide accurate and on-time support for decision making. Needed action should be clearly provided, as we cannot expect producers to hire predictors, analysts, and decision-makers for their fields. Research on real-time processing of large volumes of data combined with visualization tools providing interactive exploration is still in progress but very promising [134, 189].

Ultimately, large-scale analysis of agricultural data to support business analytics in high scale and speed necessitates investments in cloud infrastructures, while big data processing demands advanced techniques of parallel and distributed computing. Most traditional machine learning techniques are not inherently efficient or scalable to handle the challenges posed by big data in agriculture. Several current advanced learning methods, though, seem promising enough. Research funded by public organizations and work for the common good are needed to make such tools and techniques enter the public domain (and become open-sourced) [152]. Once big data research evolves and matures, it is expected that machine learning, given its history, will manage to tackle the challenges posed by big data. Successful applications of big data, though, will be determined not only by technology but by organizational and managerial factors, as well. Strong multidisciplinary engagement by producers with agricultural economists, biologists, computer scientists, and government organizations is needed to make the needed scientific advancement. Table 6.1 presents the aforementioned challenges that

arise from the use of big data analytics in agriculture and mentions the requirements needed to address them (costs).

6.3 Discussion

Challenges of agricultural production are increasing, making the need to understand the complex agricultural ecosystems more imperative than ever. Machine learning techniques are widely applied in precision agriculture due to their capabilities to mine information hidden in agricultural data. The increasing availability of data through advancements in ICT seems promising for enhancing innovation on strategic decision-making by increasing models' accuracy and generalization ability.

Without employing the data generated by precision agriculture practices, it is difficult to predict if big data will have significant impact. On the other hand, learning from massive data is expected to bring significant opportunities and transformative potential for precision agriculture. We consider precision agriculture and big data as complementary fields. Most of the cases mentioned in Sections 6.2.1.1, 6.2.1.2 and 6.2.1.3 have not been applied in farming practice yet or if they have, they have not proven their value. As the time for big data is coming, the collection and analysis of datasets is difficult to be dealt by traditional learning methods. These methods are not inherently efficient or scalable enough to work well with large-volume agricultural data exhibiting features like heterogeneity, high dimensionality and spatiotemporal autocorrelation. Several challenges on learning from big data arise but the interest to provide solutions in recent researches is apparent.

Advanced machine learning methods like convolutional neural networks offer higher accuracy, robustness, flexibility, and generalization performance. Deep neural networks seem to be the most promising technique for object recognition and classification from satellite imagery. Algorithms used to cope with such voluminous data should be easily distributed or implemented in parallel form. Big data also requires considerable technical skills to handle analysis methods, frequently demanding real-time processing, and parallel/distributed infrastructures as presented in the cost-benefit analysis of Teble 6.1.

The outlook for big data and machine learning in agriculture is very promising. High-performance scalable learning systems for data-driven discovery can turn farm management systems into artificial intelligence systems, providing richer real-time recommendations and automation of several agricultural procedures. Emerging fields of advanced machine learning and data mining combined with open datasets and policy frameworks are expected to be instrumental in helping meet the challenges of agricultural production in terms of productivity, environmental impact, food security, and sustainability.

Table 6.1: Opportunities, challenges and cost-benefit analysis of BD (analytics) adoption in agriculture

Opportunities	**Benefits**
• Characterize spatial and temporal variability in soil, crop, and environmental characteristics on precise scales • Determine growing conditions and identifying needs and threats in (near) real time • Predict yield, weather, and threats (e.g. extreme climate conditions, infections) accurately and on time • Explore hidden structures and extract common features on farms across large regions and time scales	• Automation of agricultural procedures • Accurate and timely decision making in precision agriculture • Better personalized on-farm management practices • Improved food access and supply chain management
Challenges	**Costs**
• Data quality issues	• Data quality management techniques required
• Data heterogeneity (data from multiple sources, with different formats, different time points)	• Data preparation, fusion and representation techniques required
• Data availability	• Data initiatives and producers' cultural change needed
• Data security holes and privacy concerns	• Laws and regulations needed
• Spatiotemporal autocorrelation of data	• Scalable spatiotemporal methods and explanatory space-time analysis
• High-dimensionality of data	• Spurious correlations, noise accumulation (wrong statistical inference, false conclusions, wrong discoveries) • Effective dimension reduction methods, variable selection methods and large datasets required • Combination of empirical and mechanistic models required
• Non-stationarity of data and velocity	• Stream processing and online learning techniques required • Real-time analysis frameworks for in memory computations (batch and stream) required • Combination of empirical and mechanistic models required
• Voluminous datasets	• Computational cost • Memory cost • Scalable analytics methods in parallel and distributed ways required • Parallel/distributed infrastructure in the cloud required
• Data Interpretation	• Advanced visualization techniques required • Strong multidisciplinary engagement required

Chapter 7

Conclusions

Over the last decade, the exponential growth of data from the Internet of Thing (IoT), social media, and sensing devices has introduced a massive flow of data. Real-time big data technologies are adopted in a number of application fields like agriculture due to their crucial impacts. The faster one can harness insights from data, the greater the benefit in driving value, reducing costs, and increasing efficiency.

This thesis focuses on running DSP applications in the cloud. First, popular open-source DSPSs (Chapter 2) are explored with regard to the mechanisms used to face the requirements of DSP applications. The significance of scheduling decisions on systems' performance and fault tolerance made us focus on the task placement and scheduling problem, which determine which tasks to be placed on which nodes, and control the order of task execution. Multiple stream processing computations should be interleaved on the same machine to reduce the number of needed connections and assure the necessary performance. On the other hand, heavily used machines result in memory waste, node failures and increased network congestion. Overloaded and underutilized machines should be avoided. Moreover, if streams are not managed carefully, processing delays can become unacceptable and lead to long queues at a processing node, buffer overflows, and memory exhaustion.

Several static and dynamic scheduling algorithms are investigated (Chapter 3) and, based on the aforementioned issues, a scheduling scheme is proposed and comprises the following steps: problem identification and formulation, design of resolution approaches, prototype development, and experimental evaluation.

7.1 Major Contributions

In this thesis, the task allocation and scheduling problem to handle applications that require hefty communication between nodes and tasks is investigated. Initially, a taxonomy of scheduling approaches that reveal important scheduling considerations for stream processing jobs is provided (Chapter 3). The observed parameters that affect scheduling decisions in most schemes are mainly the application's topology, the available resources, and the system's workload. Performance metrics are also important, when it

comes to online decisions.

Then attention is shifted to the formulation of the task placement and scheduling problem to provide a matrix-based approach. This approach is organized in a set of communication steps, where there is an one-to-one communication between the system's nodes and offers a set of advantages (Chapter 4):

- The buffer space required per task is reduced, resulting in higher throughput, as the largest percentage of tuples are processed as they arrive to the target node (almost no buffering is required). In case of buffered tuples resulting from each communication step, these are processed in a single step, where the pipeline is stalled. This reduces the extra processing time that would be necessary, if these tuples were processed at random times.

- The refinement phase employed by the developed strategy reduces the inter-node communication costs, by refining the task allocation to map to the specific application's DAG, so that the communicating tasks can be (to the maximum extent) placed in nearby nodes. In this way, communication latencies are reduced.

- There is almost complete load balancing in the network resulting in reduced latencies and as the scheduler itself has linear complexity, it determines the communication steps very fast.

- Finally, the scheduler is proven to be periodic, with a period equal to $LCM(t, N)$. This means, that once the first LCM(t,N) communications are arranged, the same communication pattern can follow, in case of a bigger problem, where the number of nodes or tasks is multiplied by an integer factor.

The experimental results have verified the advantages mentioned above (Chapter 5). The developed strategy offers reduced average latency and percentage of buffered memory used, compared to the default scheduler. Also, it offers good load balancing. For throughput testing, the approach presented is compared to the default scheduler as well as to R-Storm. It outperforms both the other strategies and achieves higher throughput (tuples/min) under different scenarios. The evaluation demonstrates the importance of constraining the required buffer space and achieving load balance to improve the system's performance and overcome the challenges of running DSP applications (e.g. avoid tuple losses).

One drawback of this work is that, when $G = gcd(t, N) = 1$, it needs to add a minimum number of tasks, so that the G value becomes $\neq 1$. This needs to be done before scheduling. A dynamic strategy would resolve this issue and is left for future work. A dynamic approach will model the system changes in the form of redistribution from R to R', where R is the initial task distribution between nodes and R' is the next task distribution derived from the system changes. Both R and R' will be modeled via linear Diophantine equations (which are ideal for round robin distributions) and the task redistribution will be determined by the solutions to the set of linear Diophantine equations, $R = R'$.

7.2 Future Directions

In this work, a static environment is assumed, in which bandwidth capacities and other resources do not change over time. Ingestion rates are also assumed to be static, which in reality is often not the case, due to possible changing network topologies and load fluctuations. Systems should be able to handle failures and changes of the execution environment. Continuous monitoring and adaption of schedules considerably improve solutions for real environments and applications. As shown in the literature analysis (Chapter 3) a dynamic version, where task replicas are introduced when necessary and the number of nodes change during execution, should be computed and implemented, as quickly as possible. Unfortunately, trying to maintain the necessary performance in the presence of unpredictable load variations and hardware/software performance degradation, making all the necessary reconfigurations is usually time-consuming and error-prone. Reconfigurations cause application downtime and, if applied too often, they can negatively impact the application performance.

A promising research direction regards the design of self-adaptive control policies. A key challenge here is to let the system wisely select the most profitable adaptation actions to enact. Reconfigurations should not rely on a single snapshot of the system, but should take care of the overall performance over a period of time. To address some of these challenges, machine learning techniques such as supervised learning and reinforcement learning can potentially further improve the application management lifecycle (e.g. the case of Dhalion [83]). An area that will draw the attention of many researchers in the future and may impact the ways streaming applications are managed, is the exploration of machine learning in the context of self-regulating streaming systems.

Moreover, optimizing the deployment of multiple concurrent applications is also a field for future research. An infrastructure should host multiple streaming applications, each arriving and terminating with non anticipated characteristics, such as the number of the needed nodes in the cluster or the parallelization degree of their components. In such scenarios, it is worth investigating fair ways to achieve the desired performance for each application with respect to the available resources to be shared (as scheduling defines in the introductory section "achieve minimization of task completion time and improve resource utilization"). We should always keep in mind that these requirements may be stringent and possibly conflicting but they should be met in real-time.

Finally, in this thesis, attention is paid to a specific computing resource, memory, in an attempt to reduce its utilization. Cloud computing technologies make it easy to acquire and release computing resources. Software-defined networking (SDN) technologies enable dynamic, efficient network configuration of large scale networks to improve performance and monitoring, making it more like cloud computing than traditional network management. These two technologies when combined, enable the infrastructure to be programmable, making in this way, its run-time reconfiguration able to satisfy the applications' needs. SDN technologies provide mechanisms for allocating network capacity to data flows, making it possible to determine the network paths that better suit an application's requirements (e.g. response time). Moreover, it allows to react to possible network congestion by re-routing data streams. In this way, changing the initial

task allocation or operators' replication degree will not be needed. The combination of cloud with SDN technologies rises promising in the future of data stream processing.

Appendix

In this appendix we present an example of a linear topology named *ExclamationTopology* in Apache Storm. The programming language used is Java and Maven is used to package it into a jar file, as it automates everything well.

This application has one spout (named "word") and two bolts (named "exclaim1"and "exclaim2") with 6,4, and 4 tasks respectively. There is also a dummy bolt with 10 tasks which does nothing. However, it is useful, as it provides our scheduler with the dummy tasks needed.

The spout emits words, and each bolt appends the string "!!!" to its input. The nodes are arranged in a line: the spout emits to the first bolt, which then emits to the second bolt. If the spout emits the tuples ["Hello"] and ["space"], then the third bolt will emit the words ["Hello!!!!!!"] and ["space!!!!!!"].

The nodes are defined by using the methods "setSpout" and "setBolt". These methods take as input:

- a user-specified id; "word" for the spout and "exclaim 1", "exclaim 2", and "dummy" for the bolts,
- an object containing the processing logic; TestWordSpout and ExclamationBolt extend the BaseRichSpout and the BaseRichBolt classes that implement the IRichSpout and IRichBolt interfaces accordingly. TestWordSpout emits 1-tuple every 100ms from a predefined list of entries. ExclamationBolt appends the string "!!!" to its input. It's worth mentioning that in the "execute" method of ExclamationBolt, the input tuple is passed as the first argument to "emit", and is also acked on the final line based on the Storm's reliability API for guaranteeing no data loss.
- the amount of parallelism for each component. The amount of parallelism indicates the number of threads that will execute the corresponding component across the cluster. In case it is omitted, Storm will only allocate one thread for this component.

The component "exclaim1" is set to read all the tuples emitted by the component "word" using a shuffle grouping, which means that tuples will be randomly distributed from the input tasks to the bolt's tasks. However, there are more ways to group data between components. Likewise, the "exclaim2" component will read all the tuples emitted by the component "exclaim1", using shuffle grouping, as well. The "dummy"

bolt never receives tuples from any other component. This application demands 8 workers (JVMs). In our example this means that there will be 2 JVMs per machine, since there are 4 working nodes in the cluster.

```
// ExclamationTopology.java
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 * http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.apache.storm.starter;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.testing.TestWordSpout;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;
import java.util.Map;

/**
 * This is a basic example of a Storm topology.
 */
public class ExclamationTopology4 {

  public static class ExclamationBolt extends BaseRichBolt {
    OutputCollector _collector;

    @Override
    public void prepare(Map conf, TopologyContext context,
```

```
      OutputCollector collector) {
    _collector = collector;
  }

  @Override
  public void execute(Tuple tuple) {
    _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
    _collector.ack(tuple);
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word"));
  }
 }

 public static void main(String[] args) throws Exception {
  TopologyBuilder builder = new TopologyBuilder();

  builder.setSpout("word", new TestWordSpout(), 6);
  builder.setBolt("exclaim1", new ExclamationBolt(),
      4).shuffleGrouping("word");
  builder.setBolt("exclaim2", new ExclamationBolt(),
      4).shuffleGrouping("exclaim1");
  builder.setBolt("dummy",new ExclamationBolt(), 10);

  Config conf = new Config();
  conf.setDebug(true);

  if (args != null && args.length > 0) {
    conf.setNumWorkers(8);

    StormSubmitter.submitTopology("special-topology", conf,
        builder.createTopology());
  }
  else {

    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("test", conf, builder.createTopology());
    Utils.sleep(10000);
    cluster.killTopology("test");
    cluster.shutdown();
  }
 }
}
```

Bibliography

[1] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan, "The rise of big data on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98 – 115, 2015.

[2] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, "Trends in big data analytics," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2561 – 2573, 2014, special Issue on Perspectives on Parallel and Distributed Processing.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107–113, 2008.

[4] The Apache Software Foundation. "Welcome to Apache Hadoop". Accessed: 30 September 2018. [Online]. Available: http://hadoop.apache.org

[5] M. Dias de Assuncao, R. N. Calheiros, S. Bianchi, M. A. Netto, and R. Buyya, "Big data computing and clouds: Trends and future directions," *Journal of Parallel and Distributed Computing*, vol. 79-80, pp. 3 – 15, 2015, special Issue on Scalable Systems for Big Data Management and Analytics.

[6] N. Tantalaki, S. Souravlas, and M. Roumeliotis, "A review on big data real-time stream processing and its scheduling techniques," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, no. 5, pp. 571–601, 2020.

[7] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, p. 42–47, 2005.

[8] Rajeshwari U and B. S. Babu, "Real-time credit card fraud detection using streaming analytics," in *Proceedings of the 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, pp. 439–444.

[9] The Apache Software Foundation. "Samza-What is Samza?". Accessed: 30 September 2018. [Online]. Available: http://samza.apache.org

[10] Z. Milosevic, W. Chen, A. Berry, and F. Rabhi, "Chapter 2 - real-time analytics," in *Big Data*, R. Buyya, R. N. Calheiros, and A. V. Dastjerdi, Eds. Morgan Kaufmann, 2016, pp. 39 – 61.

[11] K. Govindarajan, S. Kamburugamuve, P. Wickramasinghe, V. Abeykoon, and G. Fox, "Task scheduling in big data - review, research challenges, and prospects," in *Proceedings of the Ninth International Conference on Advanced Computing (ICoAC)*, 2017, pp. 165–173.

[12] M. Rychlý, P. Skoda, and P. Smrz, "Heterogeneity-aware scheduler for stream processing frameworks," *International Journal of Big Data Intelligence*, vol. 2, no. 2, pp. 70–80, 2015.

[13] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *Proceedings of the IEEE INFOCOM - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[14] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4334, 2017.

[15] W. Zhang, T. Rajasekaran, T. Wood, and M. Zhu, "Mimp: Deadline and interference aware scheduling of hadoop virtual machines," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 394–403.

[16] Y. Wang and W. Shi, "Budget-driven scheduling algorithms for batches of mapreduce jobs in heterogeneous clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 306–319, 2014.

[17] The Apache Software Foundation. "Apache Storm". Accessed: 30 September 2018. [Online]. Available: http://storm.apache.org/

[18] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3553–3569, 2017.

[19] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. ACM, 2015, p. 149–161.

[20] Cheng-Zhang Peng, Ze-Jun Jiang, Xiao-Bin Cai, and Zhi-Ke Zhang, "Real-time analytics processing with mapreduce," in *Proceedings of the International Conference on Machine Learning and Cybernetics*, vol. 4, 2012, pp. 1308–1311.

[21] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *Proceedings of the IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 797–802.

[22] P. Carbone, G. Gévay, G. Hermann, A. Katsifodimos, J. Soto, V. Markl, and S. Haridi, "Large-scale data stream processing systems," in *Zomaya A., Sakr S. (eds) Handbook of Big Data Technologies*. Springer, 2017.

[23] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," in *Proceedings of the 18th International Database Engineering Applications Symposium*, ser. IDEAS '14. ACM, 2014, p. 356–361.

[24] R. Ranjan, "Streaming big data processing in datacenter clouds," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 78–83, 2014.

[25] C. Georgiadis, "An evaluation and performance comparison of different approaches for data stream processing," Master's thesis, Uppsala University, Department of Information Technology, 2016.

[26] M. Dias de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1 – 17, 2018.

[27] S. Kamburugamuve and G. Fox, "Survey of distributed stream processing," 2016. [Online]. Available: www.doi.org/10.13140/RG.2.1.3856.2968

[28] M. Singh, M. A. Hoque, and S. Tarkoma, "A survey of systems for massive stream analytics," *arXiv: Distributed, Parallel, and Cluster Computing*, 2016.

[29] The Apache Software Foundation. "Apache Spark". Accessed: 9 September 2018. [Online]. Available: http://spark.apache.org

[30] The Apache Software Foundation. "Introduction to Apache Flink". Accessed: 30 September 2018. [Online]. Available: https://flink.apache.org/introduction.html

[31] S. Shahrivari, "Beyond batch processing: Towards real-time and streaming big data," *Computer Science*, vol. 3, pp. 117–129, 2014.

[32] C. Philip Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, vol. 275, pp. 314 – 347, 2014.

[33] SQLStream. Sqlstream | streaming sql analytics for kafka kinesis. Accessed: 8 August 2018. [Online]. Available: https://sqlstream.com

[34] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, p. 221–230.

[35] Amazon Web Services. "Amazon Redshift-The most popular and fastest cloud data warehouse". Accessed: 21 November 2019. [Online]. Available: https://aws.amazon.com/redshift/

[36] K. Patroumpas and T. Sellis, "Maintaining consistent results of continuous queries under diverse window specifications," *Information Systems*, vol. 36, no. 1, p. 42–61, 2011.

[37] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, p. 1792–1803, 2015.

[38] L. Affetti, R. Tommasini, A. Margara, G. Cugola, and E. D. Valle, "Defining the execution semantics of stream processing engines," *Journal of Big Data*, vol. 4, p. 12, 2017.

[39] H. Yang, "Solving problems of imperfect data streams by incremental decision trees," *Journal of Emerging Technologies in Web Intelligence*, vol. 5, 2013.

[40] V. Gudivada, A. Apon, and J. Ding, "Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations," *International Journal on Advances in Software*, vol. 10, pp. 1–20, 07 2017.

[41] J.Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, 2005, pp. 779–790.

[42] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*. Association for Computing Machinery and Morgan; Claypool, 2016.

[43] The Apache Software Foundation. "Welcome to Apache Zookeeper". Accessed: 12 February 2017. [Online]. Available: https://zookeeper.apache.org

[44] ——. "Apache Kafka-A distributed streaming platform". Accessed: 30 September 2018. [Online]. Available: https://kafka.apache.org/

[45] D. Namiot, "On big data stream processing," *International Journal of Open Information Technologies*, vol. 3, no. 8, pp. 48–51, 2015.

[46] M. Singh, M. A. Hoque, and S. Tarkoma, "A survey of systems for massive stream analytics," *arXiv: Distributed, Parallel, and Cluster Computing*, 2016. [Online]. Available: https://arxiv.org/pdf/1605.09021.pdf

[47] E. Friedman and K. Tzoumas, *Introduction to Apache Flink: Stream Processing for Real Time and Beyond*, 1st ed. O'Reilly Media, Inc., 2016.

[48] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[49] P. Córdova, "Analysis of real time stream processing systems considering latency," Data Science Association, Tech. Rep., 04 2015. [Online]. Available: http://www.datascienceassn.org/content/analysis-real-time-stream-processing-systems-considering-latency

[50] S. Hagedorn, P. Gotze, O. Saleh, and K.-U. Sattler, "Stream processing platforms for analyzing big dynamic data," *it - Information Technology*, vol. 58, no. 4, pp. 195 – 205, 2016.

[51] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, ser. HotCloud'12. USENIX Association, 2012, p. 10.

[52] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 38, pp. 28–38, 2015.

[53] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1789–1792.

[54] S. Perera, A. Perera, and K. Hakimzadeh, "Reproducible experiments for comparing apache flink and apache spark on public clouds," *ArXiv*, vol. abs/1610.04493, 2016. [Online]. Available: https://arxiv.org/pdf/1610.04493.pdf

[55] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Proceedings of the IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 2014, pp. 69–78.

[56] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *Proceedings of the IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1507–1518.

[57] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys*, vol. 44, no. 3, Jun. 2012.

[58] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[59] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. Zdonik, *Load Management and High Availability in the Borealis Distributed Stream Processing Engine*. Springer Berlin Heidelberg, 2008, pp. 66–85.

[60] R. Bhartia, "Amazon kinesis and apache storm-building a real-time sliding-window dashboard over streaming data. amazon web services," *Amazon Web Services*, 2014. [Online]. Available: https://d0.awsstatic.com/whitepapers/building-sliding-window-analysis-of-clickstream-data-kinesis.pdf

[61] IBM. "IBM-Infosphere streams". Accessed: 30 September 2017. [Online]. Available: http://www-01.ibm.com/software/data/infosphere/stream

[62] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. ACM, 2013, pp. 1–16.

[63] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: Dynamic resource scheduling for real-time analytics over fast streams," in *Proceedings of the IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 411–420.

[64] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. ACM, 2014, p. 1–13.

[65] Xinyi Liao, Zhiwei Gao, Weixing Ji, and Yizhuo Wang, "An enforcement of real time scheduling in spark streaming," in *Proceedings of the Sixth International Green and Sustainable Computing Conference (IGSC)*, 2015, pp. 1–6.

[66] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. ACM, 2017, p. 374–389.

[67] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. ACM, 2013, pp. 207–218.

[68] D. Xiang, Y. Wu, P. Shang, J. Jiang, J. Wu, and K. Yu, "RB-Storm: Resource balance scheduling in apache storm," in *Proceedings of the 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, 2017, pp. 419–423.

[69] P. Smirnov, M. Melnik, and D. Nasonov, "Performance-aware scheduling of streaming applications using genetic algorithm," *Procedia Computer Science*, vol. 108, pp. 2240–2249, 2017.

[70] L. Eskandari, Z. Huang, and D. Eyers, "P-scheduler: Adaptive hierarchical scheduling in apache storm," in *Proceedings of the Australasian Computer Science Week Multiconference*, ser. ACSW '16. ACM, 2016, pp. 26:1–26:10.

[71] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "Iterative scheduling for distributed stream processing systems," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '18. ACM, 2018, pp. 234–237.

[72] A. Shukla and Y. Simmhan, "Model-driven scheduling for distributed stream processing systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 98 – 114, 2018.

[73] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '16. ACM, 2016, pp. 69–80.

[74] A. Al-Sinayyid and M. Zhu, "Job scheduler for streaming applications in heterogeneous distributed processing systems," *The Journal of Supercomputing*, p. 20, 2020.

[75] G. Janßen, I. Verbitskiy, T. Renner, and L. Thamsen, "Scheduling stream processing tasks on geo-distributed heterogeneous resources," in *Proceedings of the IEEE International Conference on Big Data (Big Data)*, 2018, pp. 5159–5164.

[76] M. Mortazavi-Dehkordi and K. Zamanifar, "Efficient resource scheduling for the analysis of big data streams," *Intelligent Data Analysis*, vol. 23, no. 1, pp. 77–102, 2019.

[77] N. Tantalaki, S. Souravlas, M. Roumeliotis, and S. Katsavounis, "Pipeline-based linear scheduling of big data streams in the cloud," *IEEE Access*, vol. 8, pp. 117 182–117 202, 2020.

[78] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milojicic, "Adaptive scheduling of parallel jobs in spark streaming," in *Proceedings of the IEEE INFOCOM - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[79] H. Jin, F. Chen, S. Wu, Y. Yao, Z. Liu, L. Gu, and Y. Zhou, "Towards low-latency batched stream processing by pre-scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 710–722, 2019.

[80] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-aware online scheduling in Storm," in *Proceedings of the IEEE 34th International Conference on Distributed Computing Systems*, Madrid, 2014, pp. 535–544.

[81] C. Meng-Meng, Z. Chuang, L. Zhao, and X. Ke-Fu, "A task scheduling approach for real-time stream processing," in *Proceedings of the International Conference on Cloud Computing and Big Data*, Wuhan, 2014, pp. 160–167.

[82] D. Sun, H. Yan, S. Gao, X. Liu, and R. Buyya, "Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams," *Journal of Supercomputing*, vol. 74, no. 2, pp. 615–636, 2018.

[83] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in heron," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1825–1836, 2017.

[84] A. A. Safaei, "Real-time processing of streaming big data," *Real-Time Systems*, vol. 53, no. 1, p. 1–44, 2017.

[85] S. Mohammadi, "Continuous query response time improvement based on system conditions and stream features," Master's thesis, University of Science and Technology, Iran, 2010.

[86] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, Berlin, 2016, pp. 22–31.

[87] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic, "Enabling elastic stream processing in shared clusters," in *Proceedings of the IEEE 9th International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, 2016, pp. 108–115.

[88] D. Sun, S. Gao, X. Liu, F. Li, X. Zheng, and R. Buyya, "State and runtime-aware scheduling in elastic stream computing systems," *Future Generation Computer Systems*, vol. 97, pp. 194 – 209, 2019.

[89] M. Hoffmann, F. McSherry, and A. Lattuada, "Latency-conscious dataflow reconfiguration," in *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, ser. BeyondMR'18, no. 1. New York, NY, USA: ACM, 2018, pp. 1–4.

[90] B. D. Monte, "Efficient migration of very large distributed state for scalable stream processing," in *PhD@VLDB*, 2017. [Online]. Available: http://ceur-ws.org/Vol-1882/paper01.pdf

[91] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Distributed qos-aware scheduling in storm," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. New York, NY, USA: ACM, 2015, pp. 344–347.

[92] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proceedings of VLDB Endowment*, vol. 11, no. 6, pp. 705–718, 2018.

[93] G. Eisbruch, J. Leibiusky, and D. Simonassi, *Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly Media, 2012.

[94] Food and Agriculture Organization of the United Nations. (2010) "How to feed the world in 2050". Accessed: 13 February 2019. [Online]. Available: http://www.fao.org/fileadmin/templates/wsfs/docs/expert_paper/\How_to_Feed_the_World_in_2050.pdf

[95] A. Kamilaris, A. Kartakoullis, and F. X. Prenafeta-Boldú, "A review on the practice of big data analysis in agriculture," *Computers and Electronics in Agriculture*, vol. 143, pp. 23 – 37, 2017.

[96] K. Liakos, P. Busato, D. Moshou, S. Pearson, and D. Bochtis, "Machine learning in agriculture: A review." *Sensors*, vol. 18, no. 8, p. 2674, 2018.

[97] G. Morota, R. Ventura, K. M. Silva, F. F, and S. Fernando, "Big data analytics and precision animal agriculture symposium: Machine learning and data mining advance predictive big data analysis in precision animal agriculture." *Journal of Animal Science*, vol. 96, no. 4, pp. 1540–1550, 2018.

[98] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. N. Hindia, "An overview of internet of things (iot) and data analytics in agriculture: Benefits and challenges," *IEEE Internet of Things Journal*, vol. 5, no. 5, pp. 3758–3773, 2018.

[99] N. Wang and Z. Li, "8 - wireless sensor networks (wsns) in the agricultural and food industries," in *Robotics and Automation in the Food Industry*, ser. Woodhead Publishing Series in Food Science, Technology and Nutrition, D. G. Caldwell, Ed. Woodhead Publishing, 2013, pp. 171 – 199.

[100] P. P. Lerdsuwan P., "An energy-efficient transmission framework for iot monitoring systems in precision agriculture." in *ICISA 2017: Information Science and Applications 2017, Lecture Notes in Electrical Engineering, Kim K., Joukov N. (eds)*, 2017, pp. 714–721.

[101] J. Meersmans, F. De Ridder, F. Canters, S. De Baets, and M. Van Molle, "A multiple regression approach to assess the spatial distribution of soil organic carbon (soc) at the regional scale (flanders, belgium)." *Geoderma*, vol. 143, no. 1-2, pp. 1–13, 2008.

[102] S. Drummond, K. Sudduth, A. Joshi, S. Birrell, and N. Kitchen, "Statistical and neural methods for site-specific yield prediction," *Transactions of the ASAE*, vol. 6, no. 1, pp. 5–14, 2003.

[103] D. M. Lambert, J. Lowenberg-Deboer, and R. Bongiovanni, "A comparison of four spatial regression models for yield monitor data: A case study from argentina." *Precision Agriculture*, vol. 5, no. 6, pp. 579–600, 2004.

[104] E. Sadler, J. Jones, and K. Sudduth, "Modeling for precision agriculture:how good is good enough, and how can we tell?" in *Proceedings of the 6th European Conference on Precision Agriculture*, 2007, pp. 241–248.

[105] I. Ali, F. Greifeneder, J. Stamenkovic, N. Maxim, and C. Notarnicola, "Review of machine learning approaches for biomass and soil moisture retrievals from remote sensing data," *Remote Sensing*, vol. 7, no. 12, pp. 16 398–16 421, 2015.

[106] K. H. Coble, A. K. Mishra, S. Ferrell, and T. Griffin, "Big Data in agriculture: A challenge for the future," *Applied Economic Perspectives and Policy*, vol. 40, no. 1, pp. 79–96, 2018.

[107] N. Tantalaki, S. Souravlas, and M. Roumeliotis, "Data-driven decision making in precision agriculture: The rise of big data in agricultural systems," *Journal of Agricultural & Food Information*, vol. 20, no. 4, pp. 344–380, 2019.

[108] G. Ruß, "Data mining of agricultural yield data: A comparison of regression models." in *Lecture Notes in Computer Science, P. Perner (Ed.)*, 2009, pp. 24–37.

[109] H. Zheng, L. Chen, X. Han, X. Zhao, and Y. Ma, "Classification and regression tree (cart) for analysis of soybean yield variability among fields in northeast china: The importance of phosphorus application rates under drought conditions," *Agriculture, Ecosystems Environment*, vol. 132, no. 1, pp. 98 – 105, 2009.

[110] A. Gonzalez-Sanchez, J. Frausto-Solis, and W. Ojeda-Bustamante, "Predictive ability of machine learning methods for massive crop yield prediction." *Spanish Journal of Agricultural Research*, vol. 12, no. 2, pp. 313–328, 2014.

[111] X. Pantazi, D. Moshou, T. Alexandridis, R. Whetton, and A. Mouazen, "Wheat yield prediction using machine learning and advanced sensing techniques," *Computers and Electronics in Agriculture*, vol. 121, pp. 57 – 65, 2016.

[112] M. J. Diamantopoulou, "Artificial neural networks as an alternative tool in pine bark volume estimation," *Computers and Electronics in Agriculture*, vol. 48, no. 3, pp. 235 – 244, 2005.

[113] D. Tuia, J. Verrelst, L. Alonso, F. Perez-Cruz, and G. Camps-Valls, "Multioutput support vector regression for remote sensing biophysical parameter estimation," *IEEE Geoscience and Remote Sensing Letters*, vol. 8, no. 4, pp. 804–808, 2011.

[114] D. S. Kimes, R. F. Nelson, W. A. Salas, and D. L. Skole, "Mapping secondary tropical forest and forest age from spot hrv data," *International Journal of Remote Sensing*, vol. 20, no. 18, pp. 3625–3640, 1999.

[115] I. Nitze, U. Schulthess, and H. Asche, "Comparison of machine learning algorithms random forest, artificial neural network and support vector machine to maximum likelihood for supervised crop type classification." in *Proceedings of the 4th Geobia*, 2012, pp. 35–40.

[116] R. Moreno, F. Corona, A. Lendasse, M. Graña, and L. S. Galvão, "Extreme learning machines for soybean classification in remote sensing hyperspectral images," *Neurocomputing*, vol. 128, pp. 207 – 216, 2014.

[117] E. Raczko and B. Zagajewski, "Comparison of support vector machine, random forest and neural network classifiers for tree species classification on airborne hyperspectral apex images." *European Journal of Remote Sensing*, vol. 50, no. 1, pp. 144–154, 2017.

[118] F. Lahoche, C. Godard, T. Fourty, V. Lelandais, and D. Lepoutre, "An innovative approach based on neural networks for predicting soil component variability." in *Proceedings of the 6th International Conference on Precision Agriculture and Other Precision Resources Management*, Minneapolis, MN, USA, 2003, pp. 803–816.

[119] K. Were, D. Bui, and S. B. Dick, ØB, "A comparative assessment of support vector regression, artificial neural networks, and random forests for predicting and mapping soil organic carbon stocks across an afromontane landscape." *Ecological Indicators*, vol. 52, pp. 394–403, 2015.

[120] L. Pasolli, C. Notarnicola, and L. Bruzzone, "Estimating soil moisture with the support vector regression technique," *IEEE Geoscience and Remote Sensing Letters*, vol. 8, no. 6, pp. 1080–1084, 2011.

[121] O. Rahmati, H. R. Pourghasemi, and A. M. Melesse, "Application of gis-based data driven random forest and maximum entropy models for groundwater potential mapping: A case study at mehran region, iran," *CATENA*, vol. 137, pp. 360 – 372, 2016.

[122] L. Hassan-Esfahani, A. Torres-Rua, A. Jensen, and M. Mckee, "Spatial root zone soil water content estimation in agricultural lands using bayesian-based artificial neural networks and high-resolution visual, nir, and thermal imagery." *Irrigation and Drainage*, vol. 6, no. 2, pp. 273–288, 2017.

[123] D. Pokrajac and Z. Obradovic, "Neural network-based software for fertilizer optimization in precision farming," in *Proceedings of the International Joint Conference on Neural Networks*, ser. IJCNN'01, vol. 3, Washington, DC, USA, 2001, pp. 2110–2115.

[124] T. Rumpf, A.-K. Mahlein, U. Steiner, E.-C. Oerke, H.-W. Dehne, and L. Plumer, "Early detection and classification of plant diseases with support vector machines

based on hyperspectral reflectance," *Computers and Electronics in Agriculture*, vol. 74, no. 1, pp. 91 – 99, 2010.

[125] M. Jafari, S. Minaei, N. Safaie, and F. Torkamani-Azar, "Early detection and classification of powdery mildew-infected rose leaves using anfis based on extracted features of thermal images," *Infrared Physics Technology*, vol. 76, no. C, pp. 338–345, 2016.

[126] S. Cereda, "A comparison of different neural networks for agricultural image segmentation." Master's thesis, Politecnico di Milano, 2016. [Online]. Available: https://www.politesi.polimi.it/bitstream/10589/133864/3/tesi.pdf

[127] C. Andrea, B. B. Mauricio Daniel, and J. B. José Misael, "Precise weed and maize classification through convolutional neuronal networks," in *Proceedings of the IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, Salinas, CA, USA, 2017, pp. 1–6.

[128] A. I. De Castro, J. Torres-Sánchez, J. M. Peña, F. M. Jiménez-Brenes, O. Csillik, and F. López-Granados, "An automatic random forest-obia algorithm for early weed mapping between and within crop rows using uav imagery," *Remote Sensing*, vol. 10, no. 2, 2018.

[129] N. Rao, "Big data and climate smart agriculture - Review of current status and implications for agricultural research and innovation in India." in *Proceedings Indian National Science Academy*, 2018. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2979349

[130] M. Bendre, R. Thool, and V. Thool, "Big data in precision agriculture through ict: Rainfall prediction using neural network approach." in *S. Satapathy, Y. Bhatt, A. Joshi, and D. Mishra (Eds.) Proceedings of the International Congress on Information and Communication Technology.*, 2016, pp. 165–175.

[131] P. C. Deka, A. P. Patil, P. Y. Kumar, and S. R. Naganna, "Estimation of dew point temperature using svm and elm for humid and semi-arid regions of india," *ISH Journal of Hydraulic Engineering*, vol. 24, no. 2, pp. 190–197, 2018.

[132] X. Dou and Y. Yang, "Comprehensive evaluation of machine learning techniques for estimating the responses of carbon fluxes to climatic forces in different terrestrial ecosystems," *Atmosphere*, vol. 9, no. 3, p. 83, 2018.

[133] H. Guo, L. Wang, F. Chen, and D. Liang, "Scientific big data and digital earth." *Chinese Science Bulletin*, vol. 59, no. 35, p. 5066–5073, 2014.

[134] M. Chi, A. Plaza, J. A. Benediktsson, Z. Sun, J. Shen, and Y. Zhu, "Big data for remote sensing: Challenges and opportunities," *Proceedings of the IEEE*, vol. 104, no. 11, pp. 2207–2219, 2016.

[135] J. Jones, B. B. Antle, J.M, K. Boote, R. Conant, I. Foster, H. Godfray, and . T. Wheeler, "Toward a new generation of agricultural system data, models, and knowledge products: State of agricultural systems science." *Agricultural Systems*, vol. 155, pp. 269–288, 2017.

[136] A. Chlingaryan, S. Sukkarieh, and B. Whelan, "Machine learning approaches for crop yield prediction and nitrogen status estimation in precision agriculture: A review," *Computers and Electronics in Agriculture*, vol. 151, pp. 61 – 69, 2018.

[137] Y. Ma, H. Wu, L. Wang, B. Huang, R. Ranjan, A. Zomaya, and W. Jie, "Remote sensing big data computing: Challenges and opportunities." *Future Generation Computer Systems*, vol. 51, no. C, pp. 47–60, 2015.

[138] S. Shekhar, P. Schnable, D. LeBauer, K. Baylis, and K. VanderWaal, "Agriculture big data (agbd) challenges and opportunities from farm to table." 2017. [Online]. Available: https://pdfs.semanticscholar.org/c815/75e059a826f39b47367fceaac67a8f55fb07.pdf

[139] S. Wolfert, L. Ge, C. Verdouw, and M.-J. Bogaardt, "Big data in smart farming-a review." *Agricultural Systems*, vol. 153, pp. 69–80, 2017.

[140] A. Weersink, E. Fraser, D. Pannell, E. Duncan, and S. Rotz, "Opportunities and challenges for big data in agricultural and environmental analysis." *Annual Review of Resource Economics*, vol. 10, pp. 19–37, 2018.

[141] A. Lesser, "Big data and big agriculture." Analyst, Tech. Rep., 2014. [Online]. Available: https://gigaom.com/report/big-data-and-big-agriculture/

[142] NEC. "Nec and dacom collaborate on precision farming solution to maximize yields and reduce costs". Last time accessed on October 21st, 2018. [Online]. Available: https://www.nec.com/en/press/201410/global_20141023_03.html

[143] N. Shendar. (2014) "Zadara storage helps farm intelligence build petabyte-scale 'big data for crops' analytics service in the aws cloud.". Last time accessed on November 26th, 2018. [Online]. Available: https://zadara.com/blog/2014/03/20/zadara-storage-helps-farm-intelligence-build-petabyte-scale\-big-data-for-crops-analytics-service-in-the-aws-cloud/

[144] M. Carolan, "Publicising food: Big data, precision agriculture, and co-experimental techniques of addition." *Sociologia Ruralis*, vol. 57, no. 2, pp. 135–154, 2017.

[145] "Handsfreehectare". Accessed: 11 January 2019. [Online]. Available: http://www.handsfreehectare.com/

[146] G. M. Alves and P. E. Cruvinel, "Big data environment for agricultural soil analysis from ct digital images," in *Proceedings of the IEEE Tenth International*

Conference on Semantic Computing (ICSC), Laguna Hills, CA, 2016, pp. 429–431.

[147] D. Stratoulias, V. Tolpekin, R. De By, R. Zurita-Milla, V. Retsios, W. Bijker, M. Alfi Hasan, and E. Vermote, "A workflow for automated satellite image processing: From raw vhsr data to object-based spectral information for smallholder agriculture," *Remote Sensing*, vol. 9, no. 10, p. 1048, 2017.

[148] Y. Cai, K. Guan, J. Peng, S. Wang, C. Seifert, B. Wardlow, and Z. Li, "A high-performance and in-season classification system of field-level crop types using time-series landsat data and a machine learning approach." *Remote Sensing of Environment*, vol. 210, pp. 35–47, 2018.

[149] O. A. B. Penatti, K. Nogueira, and J. A. dos Santos, "Do deep features generalize from everyday objects to remote sensing and aerial scenes domains?" in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Boston, MA, 2015, pp. 44–51.

[150] Y. Wang, A. Balmos, A. Layton, S. Noel, A. Ault, J. Krogmeier, and D. Buckmaster, "An open-source infrastructure for real-time automatic agricultural machine data processing," in *Annual International Meeting. American Society of Agricultural and Biological Engineers*, 01 2017.

[151] R. K. M. Math and N. V. Dharwadkar, "Iot based low-cost weather station and monitoring system for precision agriculture in india," in *Proceedings of the 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)*, Palladam, India, 2018, pp. 81–86.

[152] I. M. Carbonell, "The ethics of big data in big agriculture." *Internet Policy Review*, vol. 5, no. 1, 2016.

[153] E. Biffis and E. Chavez, "Satellite data and machine learning for weather risk management and food security." *Risk Analysis*, vol. 37, no. 8, pp. 1508–1521, 2017.

[154] "ACREAfrica". Accessed: 11 November 2018. [Online]. Available: https://acreafrica.com

[155] K. Raju, G. Naik, R. Ramseshan, T. Pandey, P. Joshi, K. Anantha, A. Kesava Rao, and D. K. Charyulu. (2016) "Transforming weather index-based crop insurance in India: Protecting small farmers from distress. status and a way forward". Accessed: 6 December 2018. [Online]. Available: https://core.ac.uk/download/pdf/78386894.pdf

[156] N. Kshetri, "The emerging role of big data in key development issues: Opportunities, challenges, and concerns." *Big Data Society*, vol. 1, no. 2, 2014.

[157] I. Marcu, C. Voicu, A. Drăgulinescu, O. Fratu, G. Suciu, C. Balaceanu, and M. Andronache, "Overview of iot basic platforms for precision agriculture," in *Poulkov V. (eds) Future Access Enablers for Ubiquitous and Intelligent Infrastructures. FABULOUS 2019. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, vol. 283, 2019.

[158] A. Singh, N. Shukla, and N. Mishra, "Social media data analytics to improve supply chain management in food industries." *Transportation Research. Part E: Logistics and Transportation Review*, vol. 114, pp. 398–415, 2018.

[159] R. Ip, L. Ang, K. Seng, J. Broster, and J. Pratley, "Big data and machine learning for crop protection." *Computers and Electronics in Agriculture*, vol. 151, pp. 376–383, 2018.

[160] R. Lokers, R. Knapen, S. Janssen, van Randen Y., and J. Jansen, "Analysis of big data technologies for use in agro-environmental science." *Environmental Modelling Software*, vol. 84, pp. 494–504, 2016.

[161] E. Jakku, B. Taylor, A. Fleming, C. Mason, and P. Thorburn, "Big data, big trust and collaboration: Exploring the socio-technical enabling conditions for big data in the grains industry." CSIRO, Tech. Rep., 2016, eP164134.

[162] A. Fleming, E. Jakku, L. Lim-Camacho, B. Taylor, and P. Thorburn, "Is big data for big farming or for everyone? perceptions in the australian grains industry," *Agronomy for Sustainable Development*, vol. 38, no. 3, p. 24, 2018.

[163] D. Lee and J. Choi, "Learning compressive sensing models for big spatio-temporal data," in *Proceedings of the SIAM International Conference on Data Mining*, 2015, pp. 667–675.

[164] H. Zheng, J. Li, X. Feng, W. Guo, Z. Chen, and N. Xiong, "Spatial-temporal data collection with compressive sensing in mobile sensor networks." *Sensors*, vol. 17, no. 11, p. 2575, 2017.

[165] P.-Y. Chen, S. Yang, and J. McCann, "Distributed real-time anomaly detection in networked industrial sensing systems," *Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3832–3842, 2015.

[166] A. Halevy, A. Rajaraman, and J. Ordille, "Data integration: The teenage years," in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, ser. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 9–16.

[167] R. Nowak, R. Biedrzycki, and J. Misiurewicz, "Machine learning methods in data fusion systems," in *Proceedings of the 13th International Radar Symposium*, Warsaw, Poland, 2012, pp. 400–405.

[168] N. Srivastava and R. Salakhutdinov, "Multimodal learning with deep boltzmann machines," *The Journal of Machine Learning Research*, vol. 15, no. 1, p. 2949–2980, 2014.

[169] L. Zhang, Y. Xie, L. Xidao, and X. Zhang, "Multi-source heterogeneous data fusion." in *Proceedings of the International Conference on Artificial Intelligence and Big Data (ICAIBD)*, Chengdu, China, 2018, pp. 47–51.

[170] J. Woodard, "Big data and Ag-Analytics: An open source, open data platform for agricultural environmental finance, insurance, and risk." *Agricultural Finance Review*, vol. 76, no. 1, pp. 15–26, 2016.

[171] "Open Ag Data Alliance". Accessed: 5 December 2018. [Online]. Available: http://openag.io

[172] "Global Open Data for Agriculture and Nutrition-A global partnership advocating for food security". Accessed: 5 December 2018. [Online]. Available: http://godan.info

[173] M. Sykuta, "Big data in agriculture: Property rights, privacy and competition in ag data services." *International Food and Agribusiness Management Review*, vol. 19, no. A, pp. 57–74, 2016.

[174] D. P. C. Peters, K. M. Havstad, J. Cushing, C. Tweedie, O. Fuentes, and N. Villanueva-Rosales, "Harnessing the power of big data: Infusing the scientific method with machine learning to transform ecology." *Ecosphere*, vol. 5, no. 6, pp. 1–15, 2014.

[175] H. Karimi, *Big data: techniques and technologies in geoinformatics*, F. C. P. Boca Raton, Ed. Taylor Francis Group, 2014.

[176] J. Fan, F. Han, and H. Liu, "Challenges of big data analysis." *National Science Review*, vol. 1, no. 2, pp. 293–314, 2014.

[177] S. Shekhar, Z. Jiang, R. Ali, E. Eftelioglu, X. Tang, V. Gunturi, and X. Zhou, "Spatiotemporal data mining: A computational perspective." *ISPRS International Journal of Geo-Information*, vol. 4, no. 4, pp. 2306–2338, 2015.

[178] S. Li, S. Dragicevic, F. Antón Castro, M. Sester, S. Winter, A. Coltekin, and . T. Cheng, "Geospatial big data handling theory and methods: A review and research challenges." *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 115, pp. 119–133, 2016.

[179] J. Golmohammadi, Y. Xie, J. Gupta, Y. Li, J. Cai, S. Detor, and . S. Shekhar, "An introduction to spatial data mining." Department of Computer Science and Engineering University of Minnesota, Tech. Rep., 2018, TR 18-013. [Online]. Available: https://www.cs.umn.edu/sites/cs.umn.edu/files/tech_reports/18-013_0.pdf

[180] E. Y. Chang, *PSVM: Parallelizing Support Vector Machines on Distributed Computers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 213–230.

[181] A. Baldominos, E. Albacete, Y. Saez, and P. Isasi, "A scalable machine learning online service for big data real-time analysis," in *Proceedings of the IEEE Symposium on Computational Intelligence in Big Data (CIBD)*, Orlando, FL, 2014, pp. 1–8.

[182] "Mllib is Apache Spark's scalable machine learning library". Accessed: 16 January 2019. [Online]. Available: https://spark.apache.org/mllib/

[183] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[184] "Tensorflow-An end-to-end open source machine learning platform". Accessed: 16 October 16 2018. [Online]. Available: https://www.tensorflow.org/

[185] A. Roukh, F. N. Fote, S. A. Mahmoudi, and S. Mahmoudi, "Big data processing architecture for smart farming," *Procedia Computer Science*, vol. 177, pp. 78 – 85, 2020.

[186] The Apache Software Foundation. "Hbase". Accessed: 16 January 2019. [Online]. Available: https://hbase.apache.org/

[187] ——. "Cassandra-Manage massive amounts of data, fast, without losing sleep". Accessed: 16 January 2019. [Online]. Available: http://cassandra.apache.org/

[188] MongoDB. "MongoDB-The database for modern applications". Accessed: 16 January 2019. [Online]. Available: https://www.mongodb.com/

[189] M. Wachowiak, D. Walters, J. Kovacs, R. Wachowiak-Smolkov, and A. James, "Visual analytics and remote sensing imagery to support community-based research for precision agriculture in emerging areas." *Computers and Electronics in Agriculture*, vol. 143, no. C, pp. 149–164, 2017.