2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems

Scheduling Decisions in Stream Processing on Heterogeneous Clusters

Marek Rychlý
Department of Information Systems
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: rychly@fit.vutbr.cz

Petr Škoda, Pavel Šmrž
Department of Computer Graphics and Multimedia
Faculty of Information Technology, Brno University of Technology
IT4Innovations Centre of Excellence
Brno, Czech Republic
Email: {iskoda,smrz}@fit.vutbr.cz

***Abstract*—Stream processing is a paradigm evolving in response to well-known limitations of widely adopted MapReduce paradigm for big data processing, a hot topic of today's computer world. Moreover, in the field of computation facilities, heterogeneity of data processing clusters, intended or unintended, is starting to be relatively common. This paper deals with scheduling problems and decisions in stream processing on heterogeneous clusters. It brings an overview of current state of the art of stream processing on heterogeneous clusters with focus on resource allocation and scheduling. Basic scheduling decisions are discussed and demonstrated on naive scheduling of a sample application. The paper presents a proposal of a novel scheduler for stream processing frameworks on heterogeneous clusters, which employs design-time knowledge as well as benchmarking techniques to achieve optimal resource-aware deployment of applications over the clusters and eventually better overall utilization of the cluster.**

***Keywords*—scheduling; resource-awareness; benchmarking; heterogeneous clusters; stream processing; Apache Storm.**

I. Introduction

As the Internet grows bigger, the amount of data that can be gathered, stored, and processed constantly increases. Traditional approaches to processing of big data, e.g., the data of crawled documents, web request logs, etc., involves mainly batch processing techniques on very large shared clusters running in parallel across hundreds of commodity hardware nodes. For the static nature of such datasets, the batch processing appears to be a suitable technique, both in terms of data distribution and task scheduling, and distributed batch processing frameworks, e.g., the frameworks that implement the MapReduce programming paradigm [1], have proved to be very popular.

However, the traditional approaches developed for the processing of static datasets cannot provide low latency responses needed for continuous and real-time stream processing when new data is constantly arriving even as the old data is being processed. In the data stream model, some or all of the input data that are to be processed are not available in a static dataset, but rather arrive as one or more continuous data streams [2]. Traditional distributed processing frameworks like MapReduce are not well suited to process data streams due to their batch-orientation. The response times of those systems are typically greater than 30 seconds while real-time processing requires response times in the (sub)seconds range [3].

To address distributed stream processing, several platforms for data or event stream processing systems have been proposed, e.g., S4 and Storm [4], [5]. In this paper, we build upon one of these distributed stream processing platforms, namely Storm. Storm defines distributed processing in terms of streams of data messages flowing from data sources (referred to as spouts) through a directed acyclic graph (referred to as a topology) of interconnected data processors (referred to as bolts). A single Storm topology consists of spouts that inject streams of data into the topology and bolts that process and modify the data.

Contrary to the distributed batch processing approach, resource allocation and scheduling in distributed stream processing is much more difficult due to dynamic nature of input data streams. In both cases, the resource allocation deals mainly with a problem of gathering and assigning resources to the different requesters while scheduling cares about which tasks and when to place on which previously obtained resources [6].

In the case of distributed batch processing, both resources allocation and tasks scheduling can be done prior to the processing of a batch of jobs based on knowledge of data and tasks for processing and of a distributed environment. Moreover, during batch processing, required resources are often simply allocated statically from the beginning to the end of the processing.

In the case of distributed stream processing, which is typically continuous, dynamic nature of input data and unlimited processing time require dynamic allocation of shared resources and real-time scheduling of tasks based on actual intensity of input data flow, actual quality of the data, and actual workload of a distributed environment. For example, resource allocation and task scheduling in Storm involves real-time decision making considering how to replicate bolts and spread them across nodes of a cluster to achieve required scalability and fault tolerance.

This paper deals with problems of scheduling in distributed data stream processing on heterogeneous clusters. The paper is organized as follows. In Section II, stream processing on heterogeneous clusters is discussed in detail, with focus on resource allocation and task scheduling, and related work and existing approaches are analysed. In Section III, a use case of distributed stream processing is presented. Section IV deals with scheduling decisions in the use case. Based on the analysis of the scheduling decisions, Section V proposes a concept of a novel scheduling advisor for distributed stream processing on heterogeneous clusters. Since this paper presents an ongoing research, Section VI discusses future work on the scheduling advisor. Finally, Section VII provides conclusions.

978-1-4799-4325-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CISIS.2014.94

II. Stream Processing on Heterogeneous Clusters

In homogeneous computing environments, all nodes have identical performance and capacity. Resources can be allocated evenly across all available nodes and effective task scheduling is determined by quantity of the nodes, not by their individual quality. Typically, resource allocation and scheduling in the homogeneous computing environments balance of workload across all the nodes which should have identical workload.

Contrary to the homogeneous computing environments, there are different types of nodes in a heterogeneous cluster with various computing performance and capacity. High-performance nodes can complete the processing of identical data faster than low-performance nodes. Moreover, the performance of the nodes depends on the character of computation and on the character of input data. For example, graphic-intensive computations will run faster on nodes that are equipped with powerful GPUs while memory-intensive computation will run faster on nodes with large amount of RAM or disk space. To balance workload in a heterogeneous cluster optimally, a scheduler has to (1) know performance characteristics for individual types of nodes employed in the cluster for different types of computations and to (2) know or to be able to analyse computation characteristics of incoming tasks and input data.

The first requirement, i.e., the performance characteristics for individual types of employed nodes, means the awareness of infrastructure and topology of a cluster including detailed specification of its individual nodes. In the most cases, this information is provided at the cluster design-time by its administrators and architects. Moreover, the performance characteristics for individual nodes employed in a cluster can be adjusted at the cluster's run-time based on historical data of performance monitoring and their statistical analysis of processing different types of computations and data by different types of nodes.

The second requirement is the knowledge or the ability to analyse computation characteristics of incoming tasks and input data. In batch processing, tasks and data in a batch can be annotated or analysed in advance, i.e., before the batch is executed, and acquired knowledge can be utilized in optimal allocation of resources and efficient task scheduling. In stream processing, the second requirement is much more difficult to meet due to continuous flow and unpredictable variability of the input data which make thorough analysis of computation characteristics of the input data and incoming tasks impossible, especially with real-time limitations in their processing.

To address the above mentioned issues of stream processing in heterogeneous clusters with optimal performance, user-defined tasks processing (at least some) of the input data has to help the scheduler. For example, an application may include user-defined helper-tasks tagging input data at run-time by their expected computation characteristics for better scheduling[1]. Moreover, individual tasks of a stream application should be tagged at design-time according to their required computation resources and real-time constraints on the processing to help with their future scheduling. Implementation of the mentioned tagging of tasks at design-time should be part of modelling (a meta-model) of topology and infrastructure of such applications.

[1] e.g., parts of variable-bit-rate video streams with temporary high bit-rate will be tagged for processing by special nodes with powerful video decoders, while average bit-rate parts can be processed by common nodes

With the knowledge of the performance characteristics for individual types of nodes employed in a cluster and with the knowledge or the ability to analyse computation characteristics of incoming tasks and input data, a scheduler has enough information for balancing workload of the cluster nodes and optimizing throughput of an application. Related scheduling decisions, e.g., rebalancing of the workload, are usually done periodically with an optimal frequency. An intensive rebalancing of the workload across the nodes can cause high overhead while an occasional rebalancing may not utilize all nodes optimally.

A. Related Work

Over the past decade, stream processing has been the subject of a vivid research. Existing approaches can essentially be categorised by scalability into centralized, distributed, and massively-parallel stream processors. In this section, we will focus mainly on distributed and massively-parallel stream processors but also on their successors exploiting ideas of the MapReduce paradigm in the context of the stream processing.

In distributed stream processors, related work is mainly based on Aurora* [7], which has been introduced for scalable distributed processing of data streams. An Aurora* system is a set of Aurora* nodes that cooperate via an overlay network within the same administrative domain. The nodes can freely relocate load by decentralized, pairwise exchange of the Aurora stream operators. Sites running Aurora* systems from different administrative domains can by integrated into a single federated system by Medusa [7]. Borealis [8] has introduced a refined QoS optimization model for Aurora*/Medusa where the effects of load shedding on QoS can be computed at every point in the data flow, which enables better strategies for load shedding.

Massively-parallel data processing systems, in contrast to the distributed (and also centralised) stream processors, have been designed to run on and efficiently transfer large data volumes between hundreds or even thousands of nodes. Traditionally, those systems have been used to process finite blocks of data stored on distributed file systems. However, newer systems such as Dryad [9], Hyracks [10], CIEL [11], DAGuE [12], or Nephele framework [13] allow to assemble complex parallel data flow graphs and to construct pipelines between individual parts of the flow. Therefore, these parallel data flow systems in general are also suitable for the streaming applications.

The latest related work is based mainly on the MapReduce paradigm or its concepts in the context of stream processing. At first, Hadoop Online [14] extended the original Hadoop by ability to stream intermediate results from map to reduce tasks as well as the possibility to pipeline data across different MapReduce jobs. To facilitate these new features, the semantics of the classic reduce function has been extended by time-based sliding windows. Li et al. [15] picked up this idea and further improved the suitability of Hadoop-based systems for continuous streams by replacing the sort-merge implementation for partitioning by a new hash-based technique. The Muppet system [16], on the other hand, replaced the reduce function of MapReduce by a more generic and flexible update function.

S4 [4] and Apache Storm [5], which is used in this paper, can also be classified as massively-parallel data processing systems with a clear emphasis on low latency. They are not based on MapReduce but allows developers to assemble arbitrarily a

complex directed acyclic graph (DAG) of processing tasks. For example, Storm does not use intermediate queues to pass data items between tasks. Instead, data items are passed directly between the tasks using batch messages on the network level to achieve a good balance between latency and throughput.

The distributed and massively-parallel stream processors mentioned above usually do not explicitly solve adaptive resource allocation and task scheduling in heterogeneous environments. For example, in [17], authors demonstrate how Aurora*/Medusa handles time-varying load spikes and provides high availability in the face of network partitions. They concluded that Medusa with the Borealis extension does not distribute load optimally but it guarantees acceptable allocations; i.e., either no participant operates above its capacity, or, if the system as a whole is overloaded, then all participants operate at or above capacity. The similar conclusions can be done also in the case of the previously mentioned massively-parallel data processing systems. For example, DAGuE does not target heterogeneous clusters which utilize commodity hardware nodes but can handle intra-node heterogeneity of clusters of supercomputers where a runtime DAGuE scheduler decides at runtime which tasks to run on which resources [12].

Another already mentioned massively-parallel stream processing system, Dryad [9], is equipped with a robust scheduler which takes care of nodes liveness and rescheduling of failed jobs and tracks execution speed of different instances of each processor. When one of these instances underperforms the others, a new instance is scheduled in order to prevent slowdowns of the computation. Dryad scheduler works in greedy mode, it does not consider sharing of cluster among multiple systems.

Finally, in the case of approaches based on the MapReduce paradigm or its concepts, resource allocation and scheduling of stream processing on heterogeneous clusters is necessary due to utilization of commodity hardware nodes. In stream processing, data placement and distribution are given by a user-defined topology (e.g., by pipelines in Hadoop Online [14], or by a DAG of interconnected spouts and bolts in Apache Storm [5]). Therefore, approaches to the adaptive resource allocation and scheduling have to discuss initial distribution and periodic rebalancing of workload (i.e., tasks, not data) across nodes according to different processing performance and specialisation of individual nodes in a heterogeneous cluster.

For instance, S4 [4] uses Apache ZooKeeper to coordinate all operations and for communication between nodes. Initially, a user defines in ZooKeeper which nodes should be used for particular tasks of a computation. Then, S4 employs some nodes as backups for possible node failure and for load balancing.

Adaptive scheduling in Apache Storm has been addressed in [18] by two generic schedulers that adapt their behaviour according to a topology and a run-time communication pattern of an application. Experiments shown improvement in latency of event processing in comparison with the default Storm scheduler, however, the schedulers do not take into account the requirements discussed in the beginning of Section II, i.e., the explicit knowledge of performance characteristics for individual types of nodes employed in a cluster for different types of computations and the ability to analyse computation characteristics of incoming tasks and input data. By implementation of these requirements, efficiency of the scheduling can be improved.

III. Use Case

To demonstrate the scheduling problems anticipated in the current state of the art of stream processing on heterogeneous clusters, an sample application is presented in this section. The application "Popular Stories" implements a use case of processing of continuous stream of web-pages from thousands of RSS feeds. It analyses the web-pages in order to find texts and photos identifying the most popular connections between persons and related keywords and pictures. The result is a list of triples (a person's name, a list of keywords, and a set of photos) with meaning: a person recently frequently mentioned in context of the keywords (e.g., events, objects, persons, etc.) and the photos. The application holds a list of triples with the most often seen persons, keywords, and pictures for some period of time. This way, current trends of persons related to keywords with relevant photos can be obtained[2].

The application utilizes Java libraries and components from various research projects and Apache Storm as a stream processing framework including its partial integration into Apache Hadoop as a data distribution platform. Figure 1 depicts spouts and bolts components of the application and its topology, as known from Apache Storm. The components can be scaled into multiple instances and deployed on different cluster nodes.

The stream processing starts by the *URL generator* spout which extracts URLs of web-pages from RSS feeds. After that, *Downloader* gets the (X)HTML source, styles, and pictures of each web-page and encapsulates them into a stand-alone message. The message is passed to the *Analyzer bolt*, which searches the web-page for person names and for keywords and pictures in context of the names found. The resulting pairs of name-keyword are stored in tops lists in *In-memory store NK* which is updated each time a new pair arrives and excessively old pairs are removed from computation of the list. In other words, the window of a time period is held for the tops list. All changes in the tops list are passed to *In-memory store NKP*.

Moreover, pairs of name-picture emitted by *Analyzer* are processed in *Image feature extractor* to get indexable features of each image which allows later to detect different instances of the same pictures (e.g., the same photo in different resolution or with different cropping). The image features are sent to *In-memory store NP* where the tops list of the most popular persons and related unique images pairs is held. The memory stores employ search engine Apache Lucene[3] with distributed Hadoop-based storage Katta[4] for Lucene indexes to detect different instances of the same pictures as mentioned above. All modifications in the tops list of *In-memory store NP* are emitted to *In-memory store NKP* which maintains a consolidated tops list of persons with related keywords and pictures. This tops list is persistent and available for further querying.

Individual components of the application described above, both spouts and bolts, utilize various types of resources to perform various types of processing. More specifically, *URL*

[2]The application has been developed for demonstration purposes only, particularly to evaluate the scheduling advisor described in the paper. However, it may be used also in practice, e.g., for visualisation of popular news on persons (with related keywords and photos) from news feeds on the Internet.

[3]https://lucene.apache.org/

[4]http://katta.sourceforge.net/

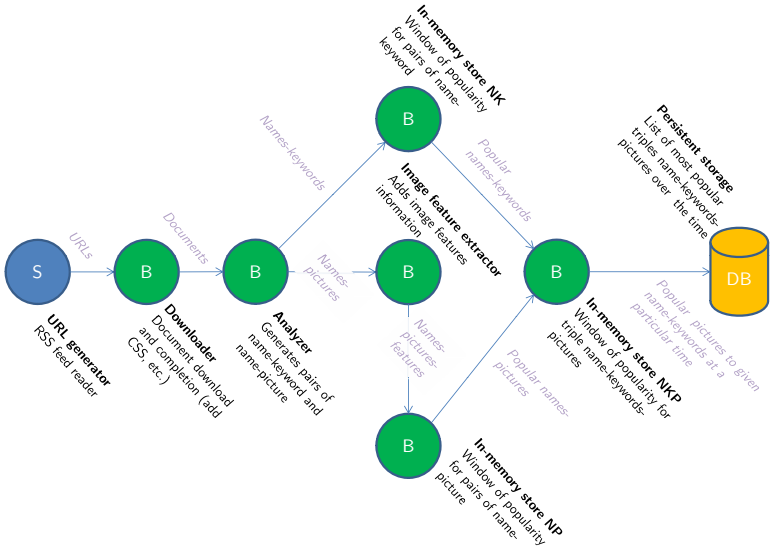

Figure 1. A Storm topology of the sample application ("S"-nodes are Storm stouts generating data and "B"-odes are Storm bolts processing the data).

generator and *Downloader* have low CPU requirements, *Analyzer* requires fast CPU, *Image feature extractor* can use GPU using OpenCL, and *In-memory stores* require large amount of memory. Therefore, the application should utilize a heterogeneous cluster with adaptive resource allocation and scheduling.

IV. Scheduling Decisions in the Stream Processing

Schedulers make their decisions on a particular level of abstraction. They do not try to schedule all live tasks to all possible cluster nodes but just deal with units of equal or scalable size. For example, the YARN scheduler uses *Containers* with various amounts of cores and memory and Apache Storm uses *Slots* of equal size (one slot per CPU core) where, in each slot, multiple spouts or blots of the same topology may run.

One of the important and commonly adopted scheduler decisions is *data locality*. For instance, the main idea of MapReduce is to perform computations by the nodes where the required data are saved to prevent intensive data loading to and removing from a cluster. Data locality decisions from the stream processing perspective are different because processors usually does not operate on data already stored in a processing cluster but rather on streams coming from remote sources. Thus, in stream processing, we consider the data locality to be an approach to minimal communication costs which results, for example, in scheduling of the most communicating processor instances together to the same node or the same rack.

The optimal placement of tasks across cluster nodes may, moreover, depend on other requirements beyond the communication costs mentioned above. Typically, we are talking about CPU performance or overall node performance that makes the processing faster. For example, the performance optimization may lie in detection of tasks which are exceedingly slow in comparison to the others with the same signature. More sophisticated approaches are based on various kinds of benchmarks performed on each node in a cluster while the placement of a task is decided with respect to its detected performance on a particular node or a class of nodes. Furthermore, the presence of resources, e.g., GPU or FPGA, can be taken into account.

There are two essential kinds of scheduling decisions: *offline decisions* and *online decisions*. The former is based on the knowledge the scheduler has before any task is placed and running. In context of stream processing, this knowledge is mostly the topology and the offline decisions can, for example, consider communication channels between nodes. Online decisions are made with information gathered during the actual execution of an application, i.e., after or during initial placement of its tasks over a cluster nodes. So the counterpart for the offline topology based communication decision is decision derived from real bandwidths required between running processor instances [18]. In effect, the most of scheduling decisions in stream processing are made online or based on historical online data.

A. Storm Default Scheduler

The Storm default scheduler uses a simple round-robin strategy. It deploys bolts and spouts (collectively called *processors*) so that each node in a topology has almost equal number of processors running in each slot even for multiple topologies running in the same cluster. When tasks are scheduled, the round-robin scheduler simply counts all available slots on each node and puts processor instances to be scheduled one at the time to each node while keeping the order of nodes constant.

In a shared heterogeneous Storm cluster running multiple topologies of different stream processing applications, the round-robin strategy may, for the sample application described in Section III, result in the scenario depicted in Figure 2. The depicted cluster consists of four nodes with different hardware configurations, i.e., fast CPU, slow CPU, lots of memory, and GPU equipped (see Figure 2), so the number of slots available at each node differs but the same portion of each node is utilized as the consequence of round-robin scheduling. Moreover, the default scheduler did not respect different requirements of processors. The *Analyzers* requiring the CPU performance were

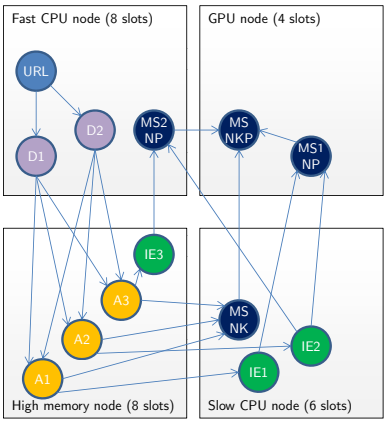

Figure 2. Possible results of the Storm default round-robin scheduler.

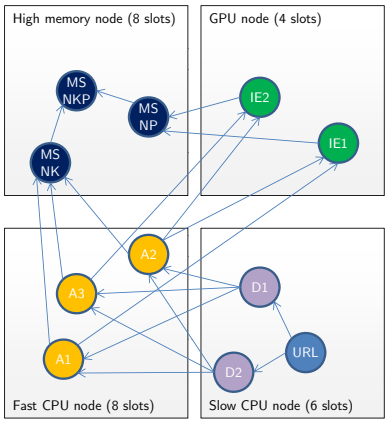

Figure 3. The advanced scheduling in a heterogeneous cluster (High memory and Fast CPU nodes are mutually swapped in comparison with Figure 2).

placed to the node with lots of memory while the memory greedy *In-memory stores* were scheduled to the nodes with powerful GPU and slow CPU which led to the need of higher level of parallelism of "MS NP". The fast CPU node then runs the undemanding *Downloaders* and the *URL generator*. Finally, the *Image extractors* were placed to the slow CPU node and the high memory node. Therefore, it is obvious that the scheduling decision was relatively wrong and it results into inefficient utilization of the cluster.

V. Proposing Scheduling Advisor

The proposed scheduling advisor targets to offline decisions derived from results of performance test sets of each resource type in combination with particular component (processor). Therefore, every application should be benchmarked on a particular cluster prior to its run in production.

The benchmarking will run the application with production-like data and after initial random or round-robin placement of processors over nodes, it will reschedule processors so that each processor is benchmarked on each class of hardware nodes. The performance of processors will be measured based on the number of tuples processed in time period. Finally, with data from benchmarks, scheduling in the production will minimize the overall counted loss of performance in deployment on particular resources in comparison to performance in the ideal deployment, i.e., the one where each processor runs on the node with top performance measured in the benchmarking phase.

Later, the scheduler can utilize also performance data captured during the production run. These data will be taken into consideration as the reflection of possible changes of processed data, and new scheduling decisions will prefer them over the performance information from the benchmarking phase. Moreover, with employment of production performance data, an application can be deployed initially using the round-robin and then gradually rescheduled in reasonable intervals. Few first reschedules have to be random to gather initial differences in performance per processor and node class. Then, the scheduler can deploy some of processors to currently best known nodes and others processors to nodes with yet unknown performance. However, when omitting the benchmarking phase, a new application without historical performance data may temporarily underperform and more instances of their processors may be needed to increase parallelism. On the other hand, without the benchmarking phase, the new application can be deployed without delay and utilize even nodes that have not yet been benchmarked (e.g., new nodes or nodes occupied by other applications during the benchmark phase on a shared cluster).

A. Scheduling of the Example Use Case Application

The proposed scheduler is trying to deploy processors to available slots that are running on nodes with the most suitable resource profile. Therefore, the scheduler will deploy fewer instance of the processors than the Storm default scheduler in the same cluster and probably even with higher throughput. In the case of the sample application, the deployment by the proposed scheduler may be as depicted in Figure 3. *In-memory stores* were deployed on the node with high amount of memory and *Image feature extractors* were deployed on the node with two GPUs so it was be possible to reduce parallelism. Undemanding *Downloaders* were placed on the Slow CPU node and *Analyzers* utilize the Fast CPU node. Possibly even more effective scheduling may be achieved by combination of pre-production and production benchmarking discussed in Section V. Then, the scheduling decisions can be based on actual bandwidths between processors with consideration of trade-offs between bandwidth availability on particular nodes shared among multiple applications and availability of more suitable nodes in perspective of performance.

VI. Discussion and Future Work

Since this paper presents ongoing work, in this section, we discuss preliminary results and outline possible further improvements. Ongoing work mainly deals with three topics: implementation of the scheduling advisor and the sample application, evaluation of the proposed approach, and its improvement based on results of the evaluation.

At the time of writing this paper, the implementation of the application described in the paper was in progress and the

scheduling advisor was in the phase of design. The scheduling advisor will be realised as a Storm scheduler implementing the IScheduler interface provided by Storm API. Besides the functionality available via Storm API, the scheduler will utilize the Apache Ambari project to monitor the Hadoop platform where a Storm cluster and other services will be running (e.g., Zookeeper coordinating various daemons within the Storm cluster, YARN managing resources of the cluster, or Hive distributed storage of large datasets generated by performance, workload, and configuration monitoring of the cluster). Apache Ambari can provide the scheduler with information on status of host systems and of individual services that run on them, including status of individual jobs running on those services.

After prototype implementation of the application and the scheduling advisor, we plan to perform thorough evaluation, to determine performance boost and workload distribution statistics and to compare these values for the scheduling advisor, the default Storm scheduler, and for generic schedulers proposed in [18]. The evaluation will also include performance monitoring and analysis of the overhead introduced by the schedulers (e.g., by monitoring and by reallocation of resources or rescheduling of processors), both per node and for the whole Storm cluster.

Finally, there are several possible improvements and open issues which we have yet to be addressed. These are ranging from improvement of scheduling algorithm performance, which is important for real-time processing (e.g., by using a hidden Markov chain based prediction algorithm for predicting input stream intensity and characteristics), through problems connected with automatic scaling of components (elasticity), to the issue of total decentralisation of the scheduler and all its components, which will be "a single point of failure" otherwise.

VII. Conclusion

This paper described problems of adaptive scheduling of stream processing applications on heterogeneous clusters and presented ongoing research towards the novel scheduling advisor. In the paper, we outlined general requirements to the scheduling in stream processing on heterogeneous clusters and analysed the state-of-the-art approaches introduced in the related works. We also described the sample application of stream processing in heterogeneous clusters, analysed scheduling decisions, and proposed the novel scheduler for the Apache Storm distributed stream processing platform based on the knowledge acquired in the previous phases.

The sample application and the proposed scheduler are still work-in-progress. We are currently implementing the application and a first prototype of the scheduler to be able to perform an evaluation of the proposed approach in practice. Our future work mainly aims at possible improvements of the scheduler performance, which is important for real-time processing, at addressing the problems connected with automatic scaling of processing components (i.e., their elasticity), and at addressing the issues related to eventual decentralisation of the scheduler implementation.

Acknowledgement

This work was supported by the BUT FIT grant FIT-S-14-2299, the European Regional Development Fund in the project CZ.1.05/1.1.00/02.0070 "The IT4Innovations Centre of Excellence", and by the EU 7FP ICT project no. 318763 "Java Platform For High Performance And Real-Time Large Scale Data Management" (JUNIPER).

References

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

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2002, pp. 1–16.

[3] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer, "Scalable and low-latency data processing with stream mapreduce," in 2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2011, pp. 48–58.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in 2010 IEEE International Conference on Data Mining Workshops (ICDMW). IEEE, 2010, pp. 170–177.

[5] N. Marz, "Apache Storm," https://git-wip-us.apache.org/repos/asf?p=incubator-storm.git, 2014, Git repository.

[6] V. Vinothina, S. Rajagopal, and P. Ganapathi, "A survey on resource allocation strategies in cloud computing," International Journal of Advanced Computer Science and Applications, vol. 3, no. 6, 2012.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing," in CIDR, vol. 3, 2003, pp. 257–268.

[8] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, , and S. Zdonik, "The design of the borealis stream processing engine," in CIDR, vol. 5, 2005, pp. 277–289.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," ACM SIGOPS Operating Systems Review, vol. 41, no. 3, pp. 59–72, 2007.

[10] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in 2011 IEEE 27th International Conference on Data Engineering (ICDE). IEEE, 2011, pp. 1151–1162.

[11] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: a universal execution engine for distributed data-flow computing," in Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, 2011.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed dag engine for high performance computing," Parallel Computing, vol. 38, no. 1, pp. 37–51, 2012.

[13] D. Warneke and O. Kao, "Exploiting dynamic resource allocation for efficient parallel data processing in the cloud," IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 6, pp. 985–997, 2011.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, 2010.

[15] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using MapReduce," in Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011, pp. 985–996.

[16] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: MapReduce-style processing of fast data," Proceedings of the VLDB Endowment, vol. 5, no. 12, pp. 1814–1825, 2012.

[17] M. Balazinska, H. Balakrishnan, and M. Stonebraker, "Load management and high availability in the Medusa distributed stream processing system," in Proceedings of the 2004 ACM SIGMOD international conference on Management of data. ACM, 2004, pp. 929–930.

[18] 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.