Heterogeneity-aware scheduler for stream processing frameworks

Marek Rychlý*

Department of Information Systems,
Faculty of Information Technology,
Brno University of Technology,
Brno, Czech Republic
Email: rychly@fit.vutbr.cz
*Corresponding author

Petr Škoda and Pavel Smrž

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

Abstract: This article discusses problems and decisions related to scheduling of stream processing applications in heterogeneous clusters. An overview of the current state of the art of the stream processing on heterogeneous clusters with a focus on resource allocation and scheduling is presented first. Then, common scheduling approaches of various stream processing frameworks are discussed and their limited applicability in the heterogeneous environment is demonstrated on a simple stream application. Finally, the article presents a novel heterogeneity-aware scheduler for the stream processing frameworks based on design-time knowledge as well as benchmarking techniques. It is shown that the scheduler overcomes alternatives in resource-aware deployment over cluster nodes and thus it leads to a better utilisation of the clusters.

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

Reference to this paper should be made as follows: Rychlý, M., Škoda, P. and Smrž, P. (2015) ‘Heterogeneity-aware scheduler for stream processing frameworks’, *Int. J. Big Data Intelligence*, Vol. 2, No. 2, pp.70–80.

Biographical notes: Marek Rychlý is an Assistant Professor at Brno University of Technology, Faculty of Information Technology (BUT FIT). His research interests are in the area of software architecture and include dynamic reconfiguration and component mobility in component-based and service-oriented architectures, formal description of software architectures and their evolution, functional and quality-driven automatic web services composition and testing, and on distributed software systems. He authored more than 20 papers in scientific journals and in conference proceedings on varied topics related to software engineering and software architectures.

Petr Škoda is PhD student and junior researcher at BUT FIT. He is responsible for the design and development of a large-scale applications based on the Hadoop and Storm frameworks. His research further tackles resource allocation and scheduling, distributed computing, and architecture of software systems. He authored three papers in scientific conference proceedings.

Pavel Smrž is an Associate Professor at BUT FIT. He leads the Knowledge Technology Research Group that has participated in many European as well as national research and development projects. His research interests include semantic technologies, large-scale distributed systems, deep learning, hardware-accelerated computing, natural language processing and human machine interaction. He has authored more than 60 papers in scientific journals and in conference proceedings.

This paper is a revised and expanded version of a paper entitled ‘Scheduling decisions in stream processing on heterogeneous clusters’ presented at the Eighth International Conference on Complex, Intelligent, and Software Intensive Systems, Birmingham, GB, 2–4 July 2014.

1 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 the batch processing techniques on very large shared clusters running in parallel across hundreds of commodity hardware nodes. Considering the static nature of such datasets, the batch processing appears to be a suitable technique both in terms of the data distribution and of the task scheduling. Consequently, the distributed batch processing frameworks, e.g., the frameworks that implement the MapReduce programming paradigm by Dean and Ghemawat (2008), have proved to be very popular.

However, the traditional approaches developed for the processing of static datasets cannot provide the low latency responses, which are needed for the continuous and real-time stream processing when the 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 to be processed are not available in a static dataset but rather they arrive as one or more continuous data streams, as described by Babcock et al. (2002). The traditional distributed processing frameworks like MapReduce are not well suited to process the data streams due to their batch orientation. The response times of those systems are typically greater than 30 seconds while the real-time processing requires the response times in the (sub)seconds range, as noted by Brito et al. (2011).

To address the distributed stream processing, several platforms for data or event stream processing systems have been proposed, e.g., S4 by Neumeyer et al. (2010) and Storm by Marz (2014). In this article, we build upon one of these distributed stream processing platforms, namely Storm. Storm defines a distributed processing in terms of streams of data messages flowing from the data sources (referred to as spouts) through a directed acyclic graph (DAG) (referred to as a topology) of the interconnected data processors (referred to as bolts). A single Storm topology consists of the spouts that inject streams of data into the topology and the bolts that process and modify the data.

Contrary to the distributed batch processing approach, the resource allocation and the scheduling in the distributed stream processing is much more difficult due to the dynamic nature of the 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, as described by Vinothina et al. (2012).

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

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

This article deals with problems of scheduling in distributed data stream processing on heterogeneous clusters. The article is organised as follows. In Section 2, the stream processing on heterogeneous clusters is discussed in detail with focus on the resource allocation and the task scheduling, along with that, the related work and the existing approaches are analysed. In Section 3, a use case of distributed stream processing is presented. Section 4 deals with scheduling decisions in the use case. Based on the analysis of the scheduling decisions, Section 5 proposes a concept of a novel scheduling advisor for the distributed stream processing on heterogeneous clusters. The implementation and evaluation of the proposed scheduling advisor is described in Section 6. Furthermore, Section 7 describes possible utilisation of the scheduling advisor in development process of applications for distributed stream processing. Finally, Section 8 outlines future work on the scheduling advisor and provides conclusions of the article.

2 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, the resource allocation and the scheduling in the homogeneous computing environments balances workload across all the nodes which results in the identical workload on each particular node.

Contrary to the homogeneous computing environments, there are different types of nodes with various computing performance and capacity in a heterogeneous cluster. The high-performance nodes then can complete the processing of identical data faster than the low-performance ones. 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 the performance characteristics of the individual types of nodes employed in the cluster for different types of computations

2 know or to be able to analyse the 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 the infrastructure and the topology of a cluster including a detailed specification of its individual nodes. In most cases, this information is provided at cluster design-time by its administrators and architects. Moreover, the performance characteristics of individual nodes employed in a cluster can be adjusted at the cluster's run-time based on the historical data of the cluster's performance monitoring and the statistical analysis of different combinations of the computations, the data, and the types of nodes.

The second requirement is the knowledge or the ability to analyse the 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 the knowledge acquired this way can be utilised to find the near optimal allocation of the resources and to find the efficient task scheduling. In the stream processing, the second requirement is much more difficult to meet due to continuous flow and unpredictable variability of the input data. These make the thorough analysis of the computation characteristics for the input data and the incoming tasks impossible especially with the real-time limitations in their processing.

To address the above mentioned issues of the stream processing in the heterogeneous clusters with optimal performance, user-defined tasks that process (at least some of) the input data have to help the scheduler with its work. For example, to achieve a better scheduling, an application may include some user-defined helper-tasks tagging the input data at run-time by their expected computation characteristics (such as tagging parts of variable-bit-rate video streams with temporary high bit-rate for processing by special nodes with powerful video decoders, while average bit-rate parts can be processed by common nodes). Moreover, individual tasks of a stream application should be tagged according to their required computation resources and real-time constraints on the processing at the design-time to help with their future scheduling. The implementation of the mentioned design-time task tagging should be a part of the modelling (a meta-model) of the topology and the infrastructure of such applications.

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

2.1 Related work

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

In the distributed stream processors, the related work is mainly based on Aurora*, which has been introduced by Cherniack et al. (2003) to address the 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 the load by decentralised, pairwise exchange of the Aurora stream operators. Sites running the Aurora* systems from different administrative domains can be integrated into a single federated system by Medusa by Cherniack et al. (2003). Abadi et al. (2005) introduced a refined QoS optimisation model for Aurora*/Medusa where the effects of the load shedding on QoS can be computed at every point in the data flow, which enables better strategies for the 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 by Isard et al. (2007), Hyracks by Borkar et al. (2011), CIEL by Murray et al. (2011), DAGuE by Bosilca et al. (2012), or the Nephele framework by Warneke and Kao (2011) 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, Condie et al. (2010) extended the original Hadoop to Hadoop Online by ability to stream intermediate results from the map tasks to the reduce tasks as well as the possibility to pipeline the 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. (2011) 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. Finally, in the Muppet system by Lam et al. (2012), the reduce function of MapReduce was replaced by a more generic and flexible update function.

S4 by Neumeyer et al. (2010) and Apache Storm by Marz (2014), which is used in this article, can also be classified as massively-parallel data processing systems with a clear emphasis on a low latency. These systems are not based on MapReduce but allow developers to assemble an arbitrarily complex DAG of processing tasks. For example, Storm does not use the intermediate queues to pass the data items between tasks. Instead, the data items are passed directly between the tasks using batch messages on the network level to achieve a good balance between the latency and the 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, Balazinska et al. (2004) analysed 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 the 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 the heterogeneous clusters which utilise the commodity hardware nodes but can handle an intra-node heterogeneity of clusters of supercomputers where DAGuE scheduler decides at runtime which tasks to run on which resources, as described by Bosilca et al. (2012).

Another already mentioned massively-parallel stream processing system, Dryad by Isard et al. (2007), is equipped with a robust scheduler which takes care of the nodes' liveness, the rescheduling of failed jobs, and tracks the execution speed of different instances of each processor. When one of these instances under-performs the others, a new instance is scheduled in order to prevent slowdowns of the computation. Dryad scheduler works in greedy mode, i.e., it does not consider sharing of the cluster among multiple systems.

Finally, in case of approaches based on the MapReduce paradigm or its concepts, resource allocation and scheduling of the stream processing on the heterogeneous clusters is necessary due to utilisation of the commodity hardware nodes. In the stream processing, data placement and distribution are given by a user-defined topology [e.g., by pipelines in Hadoop Online by Condie et al. (2010), or by a DAG of interconnected spouts and bolts in Apache Storm by Marz (2014)]. Therefore, the approaches to the adaptive resource allocation and scheduling have to discuss an initial distribution and a periodic rebalancing of a workload (i.e., tasks, not data) across the nodes according to the different processing performance and specialisation of the individual nodes in a heterogeneous cluster.

For instance, S4 by Neumeyer et al. (2010) uses Apache ZooKeeper to coordinate all operations and for the communication between the nodes. Initially, a user defines in ZooKeeper the nodes that should be used for the particular tasks of a computation. Then, S4 employs additional nodes as the backups for possible node failures and for load balancing.

An adaptive scheduling in Apache Storm has been addressed by Aniello et al. (2013). They proposed two generic schedulers that adapt their behaviour according to a topology and a run-time communication pattern of an application. Experiments shown an improvement in latency of the event processing in comparison to the default Storm scheduler. However, the proposed schedulers do not take into account the requirements discussed in the beginning of Section 2, 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 the computation characteristics of incoming tasks and input data. By implementation of these requirements, the efficiency of the scheduling can be improved.

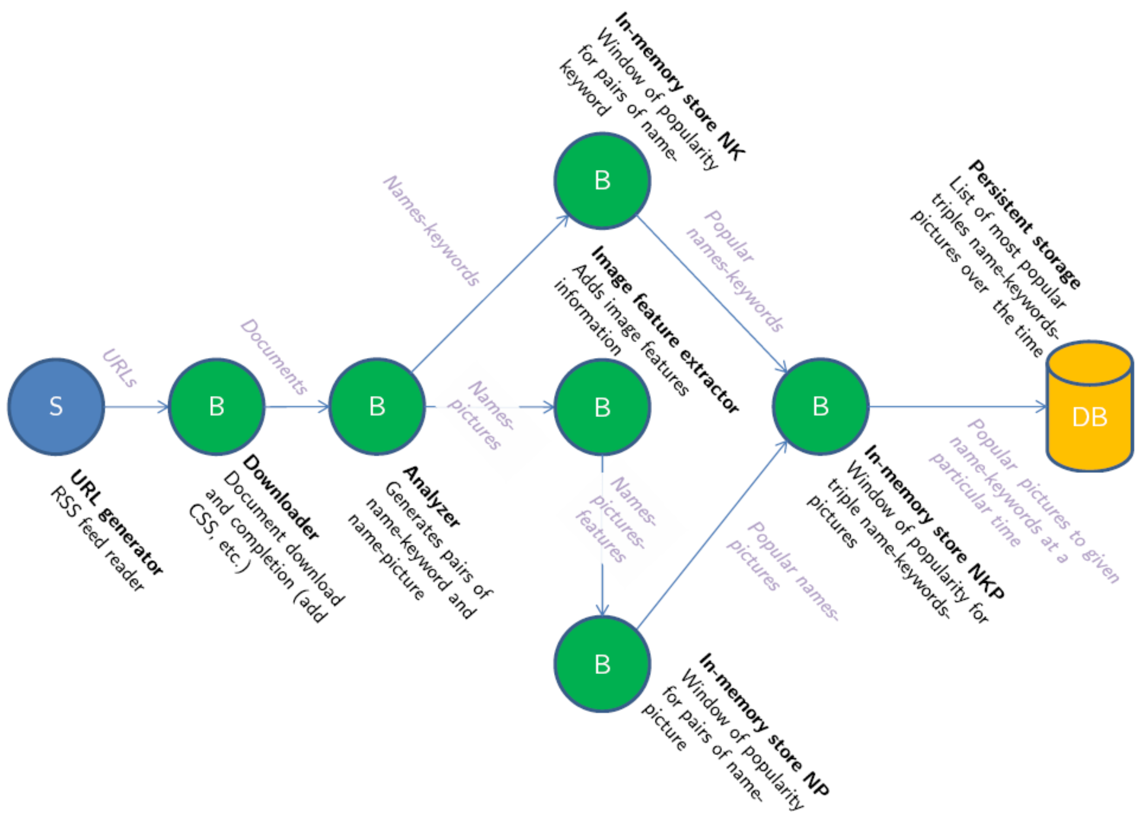
3 Use case

To demonstrate the scheduling problems anticipated in the current state of the art of the stream processing on the heterogeneous clusters, a sample application is presented in this section. The application 'popular stories' implements a use case of the processing of continuous stream of web-pages from thousands of RSS feeds. It analyses the web-pages in order to find the 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 frequently mentioned in the context of the keywords (e.g., events, objects, persons, etc.) and the photos in the recent time. The application holds a list of triples with the most often seen persons, keywords, and pictures in context of some period of time. This way, current trends of the persons related to the keywords with the relevant photos can be obtained.

The application utilises Java libraries and components from various research projects and Apache Storm as the stream processing framework. Figure 1 depicts the 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, the *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 *analyser bolt*, which searches the web-page for person names and for keywords and pictures in context of the names found before. The resulting pairs of the name-keyword are stored in the tops list in the *in-memory store NK*, which is updated each time a new pair arrives, at the same time, excessively old pairs are removed from the 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 the *in-memory store NKP*.

Figure 1 A Storm topology of the sample application (see online version for colours)


S
URL generator
RSS feed reader
URLs
B
Downloader
Document download and completion (add CSS, etc.)
Documents
B
Analyzer
Generates pairs of name-keyword and name-picture
Names-keywords
Names-pictures
B
In-memory store NK
Window of popularity for pairs of name-keyword
B
Image feature extractor
Adds image features information
Names-pictures-features
B
In-memory store NP
Window of popularity for pairs of name-picture
Popular names-keywords
Popular names-pictures
B
In-memory store NKP
Window of popularity for triple name-keywords-pictures
Popular pictures to given name-keywords at a particular time
DB
Persistent storage
List of most popular triples name-keywords-pictures over the time

Notes: 'S' – nodes are the Storm spouts generating data and 'B' – nodes are the Storm bolts processing the data.

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

The individual components of the application described above, both the spouts and the bolts, utilise the various types of resources to perform the various types of processing. More specifically, the *URL generator* and the *downloader* have low CPU requirements, the *analyser* requires fast CPU, the *image feature extractor* can employ a GPU using the OpenCL, and all the *in-memory stores* require a large amount of memory. Therefore, the application may utilise a heterogeneous cluster with an adaptive resource allocation and scheduling.

4 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 they 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 the computations by the nodes where the required data are saved both to prevent intensive data loading to a cluster before the computations and to prevent removing of the data from the cluster after the computations. The data locality decisions from the stream processing perspective are different because the processors usually do not operate on a data already stored in a processing cluster but rather on the streams coming from remote sources. Thus, in the 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 consider the CPU performance or the overall node performance that makes the processing faster. For example, the performance optimisation 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 some kinds of resources, e.g., GPU or FPGA, can be taken into account.

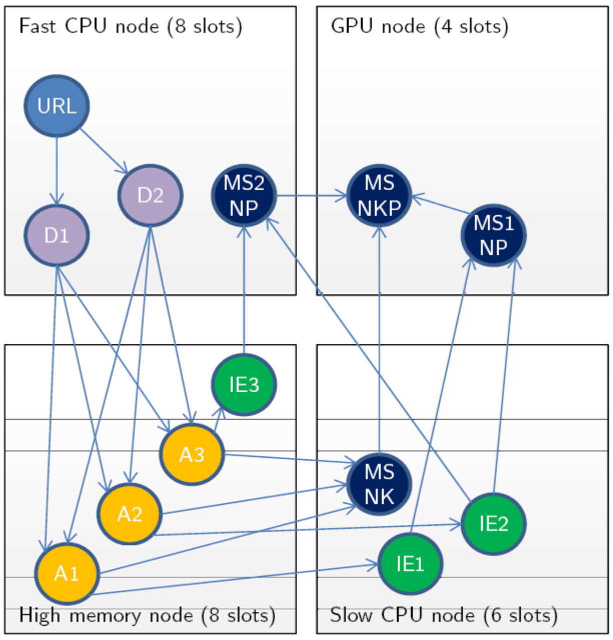
There are two essential kinds of the scheduling decisions: *offline decisions* and *online decisions*. The former is based on the knowledge the scheduler has before any task is placed and running. In context of stream processing, this knowledge is mostly the topology and the offline decisions can, for 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 the initial placement of its tasks over a cluster nodes. So the counterpart for the offline topology-based communication decision is a decision derived from the real bandwidths required between the running processor instances, as described by Aniello et al. (2013). In effect, the most of the scheduling decisions in the stream processing are made online or based on the historical online data.

4.1 *Storm's default scheduler*

The Storm's 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 the equal number of processors running in each slot even for the multiple topologies sharing the same cluster. When tasks are scheduled, the round-robin scheduler simply counts all available slots on each node and puts the 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 3, result in the scenario depicted in Figure 2. The portrayed cluster consists of four nodes with the 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 (the same number of slots) of each node is utilised as the consequence of the round-robin scheduling. Moreover, the default scheduler did not respect different requirements of the processors. The *analysers* requiring the CPU performance were placed to the node with lots of memory while the memory greedy *in-memory stores* were scheduled to the nodes with the powerful GPU and the 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 utilisation of the cluster.

Figure 2 Possible results of the Storm default round-robin scheduler (see online version for colours)



Notes: URL – URL generator; Dx – downloader; Ax – analyser; IEx – image feature extractor; MSx – in-memory store

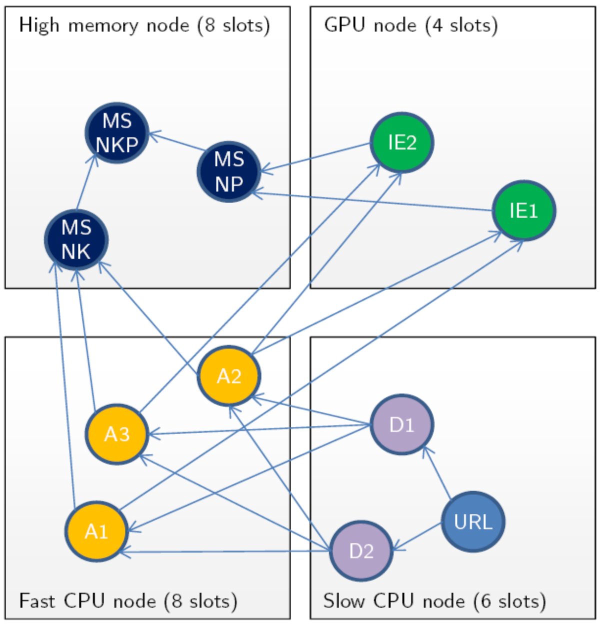
5 Proposed scheduling advisor

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

The benchmarking will run the application with production-like data and after the initial random or round-robin placement of processors over the nodes, it will reschedule the 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 a time period. Finally, with the data from the benchmarks, scheduling in the production will minimise the overall calculated loss of performance in the deployment on the particular resources in comparison to the performance in the ideal deployment, i.e., the one where each processor runs on the node with its top performance measured in the benchmarking phase.

Later, the scheduler can also utilise the performance data captured during the production run. These data will be taken into consideration as the reflection of the possible changes of processed data, and the new scheduling decisions will (in some situations) prefer them over the performance information from the benchmarking phase. Moreover, with employment of the production performance data, an application can be deployed initially using the round-robin and then gradually rescheduled in reasonable intervals. The first few reschedules have to be random to gather initial differences in the performance per processor and node class. Then, the scheduler can deploy some of the processors to currently best known nodes and other processors to nodes with yet unknown performance. However, when omitting the benchmarking phase, a new application without the historical performance data may temporarily under-perform and more instances of its processors may be needed to increase the degree of parallelism. On the other hand, without the benchmarking phase, the new application can be deployed with no delays and can utilise even the nodes that have not yet been benchmarked (e.g., the new nodes or the nodes occupied by other applications during the benchmark phase on a shared cluster).

Figure 3 The advanced scheduling in a heterogeneous cluster (see online version for colours)



Notes: High memory and Fast CPU nodes are mutually swapped in comparison with Figure 2.
URL – URL generator; Dx – downloader;
Ax – analyser; IEx – image feature extractor;
MSx – in-memory store

5.1 *Scheduling of the example use case application*

The proposed scheduler is trying to deploy the processors to the available slots that are running on nodes with the most suitable resource profile. Therefore, the scheduler may possibly deploy fewer instances of the processors than the Storm's 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 look like the one depicted in Figure 3. The *in-memory stores* were deployed on the node with a high amount of memory and the *image feature extractors* were deployed on the node with the two GPUs so it was possible to reduce the parallelism of the bolt. The undemanding *downloaders* were placed on the slow CPU node and the *analysers* utilise the fast CPU node. Possibly even more effective scheduling may be achieved by the combination of pre-production and production benchmarking discussed in Section 5. Then, the scheduling decisions can be based on the actual bandwidths between the processors with the consideration of the trade-offs between the bandwidth availability on particular nodes shared among the multiple applications and the availability of more suitable nodes in the perspective of performance.

6 Implementation and evaluation of the scheduling advisor

The scheduling advisor has been developed within the JUNIPER project as a part of a Java platform supporting the high-performance applications for the real-time access and processing of the streaming and stored data.

The scheduling advisor consists of two main components (see Figure 4): the macroscheduling component and the advisor component. The *macro-scheduling component* takes care of the scheduling decisions made on the particular hardware platform in the production or pre-production deployment of a JUNIPER application. The *advisor component*, on the other hand, analyses the performance data gathered during the production or pre-production deployment, combines them with the information from the modelling provided by developers, and shows possible shortcomings in the application design. As this article deals primarily with the scheduling, the advisor component mentioned above will be omitted and the rest of this section discusses only the scheduling component.

6.1 *The macro-scheduling component*

The macro-scheduling component is further divided into three subcomponents (see Figure 4): the monitoring component, the analysis component, and the scheduling component.

- *The monitoring component* gathers data about the performance of the individual instances of application components deployed on various hardware configurations. More precisely, it traces the execution times of program instances, which is the most important metric for the scheduling advisor prototype.
- *The analysis component* computes the performance characteristics of application components running on individual hardware classes (i.e., pairs [component, HWclass]) based on the data gathered by the monitoring component. Along with that, this component produces the first output of the scheduling advisor, namely the profiling results of an application and the benchmarking results for its individual components for various deployments of the application.
- *The scheduling component* utilises the data from the analysis component and prepares new deployments of the application components over the hardware platform

to either provide more data for the analysis component or to improve the overall performance of the application as a whole. The scheduling component produces the second output of the scheduling advisor, the best possible deployment of the application on a particular platform.

The macro-scheduling component of the scheduling advisor prototype consumes three inputs in different sub-components. The first input is a deployment package of a particular implementation of a JUNIPER application and it is utilised by the scheduling component, which takes care of actual deployment of the application components over the JUNIPER platform. The second input is a description of a particular hardware platform, which is employed by the scheduling component and the analysis component. These two components need to know the hardware classes of the nodes in the hardware platform and the counts of the nodes belonging to particular hardware classes to correctly observe and use the performance data of the different application components. The third input is a defined degree of parallelism of each application component and it is utilised by the scheduler component to correctly deploy the JUNIPER application.

6.2 *Evaluation of the scheduling advisor prototype*

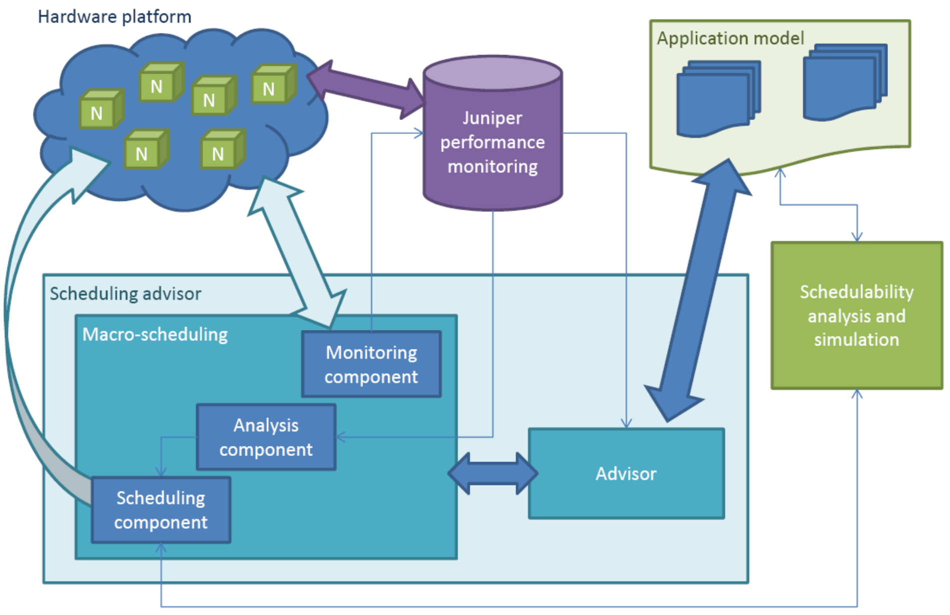
To evaluate the concept of the scheduling advisor described in this article, the prototype has been implemented as a pluggable scheduler for Apache Storm. Apache Storm was chosen as a substrate to the prototype of scheduling advisor because its model of computation is a subset of JUNIPER platform's computation model with focus on the stream processing and the low latency, and, at the same time, it offers the way to easily implement our example application 'popular stories' described in Section 3.

Different schedulers make their decisions on specific level of abstraction (i.e., they do not schedule tasks to the computation node as a whole), usually a kind of units with equal or scalable size are used. Apache Storm uses the equal size slots (one slot per a CPU core), as it was mentioned in Section 4. In each slot, multiple executors (i.e., multiple components of the JUNIPER applications) of the same topology may run. Storm's default scheduler then, using the round-robin strategy, deploys the executors in the way that each node in a topology has almost the same number of the executors running in each slot. The rule of almost the same number of the executors is maintained even when multiple topologies are running the same cluster.

The scheduling advisor prototype is partly implemented inside of Apache Storm (scheduling component and analysis component) and partly inside of the example application itself (monitoring component). The monitoring data produced by each application component instance is currently saved in the centralised relational database, which brings the possibility of the easy statistical questioning over the data. As the centralised database is not suitable for the distributed environments, later, in the future versions of the Scheduling Advisor, the monitoring data will be stored together with the rest of the performance data gathered from the platform in the JUNIPER platform's monitoring system.

The scheduling component has straightforward access to the Storm's APIs for the executor placement, removal and status so the scheduling can be easily decided by the executor type and host-name (host-names are mapped to the hardware classes using the hardware platform description). The analysis component then operates inside the Storm cluster's supervisor system called Nimbus, the same place where the scheduler resides. The analysis component questions the database with the monitoring data and provides an API to the scheduler component to pass the per-hardware-class and application component performance data and known placements.

Figure 4 Architecture of the scheduling advisor (see online version for colours)



The prototype implementation of the scheduling advisor is suitable for experiments with the heterogeneous clusters in the meaning of different hardware used over the cluster nodes (e.g., different CPUs, GPUs, amount of memory or static acceleration). It allows to periodically reschedule the application over the heterogeneous cluster and to observe the performance of the application components on the different hardware classes.

Our experiments were made on a small cluster of seven machines with three different hardware classes. Two hardware classes are of the same CPU generations, namely 'class 1' is Intel Xeon (12 cores, 3 nodes) and 'class 2' is Intel i7 (8 cores, 2 nodes). The last 'class 3' is a three years old Intel Xeon (12 cores, 2 nodes). Other parameters of the hardware classes such as amount of RAM or presence of GPU/FPGA are not important because the testing application currently does not employ them. The number of the Storm slots was set up based on the number of cores of each machine. We used multiple configurations of cluster with different numbers of machines of each class during the experiments. Our example application 'popular stories' with six different components in various degree of parallelism served as a test suite.

Different components of 'popular stories' application have different demands on performance and process different amounts of tuples over time so we evaluated the performance improvement in two ways:

1 based on the number of tuples computed by a whole application in the time interval

2 based on the number of tuples computed by each component in the time interval.

The performance is compared between the worst possible schedule, the standard schedule, and the best schedule where the worst and the best possible schedules are based on the profiling and benchmarking made by the scheduling advisor prototype and a standard schedule is made by the Storm's 'Even scheduler'.

Results of experiments showed that the performance gain of the best scheduling based on the profiling and benchmarking depends on various factors where the most important one is the structure of the heterogeneous cluster. On a homogeneous cluster, all three scheduling techniques would give almost the same results because the worst scheduling nor the best scheduling can utilise the differences of the hardware classes. Having a cluster with a small amount of 'slower' nodes brings some difference between the scheduling techniques but the difference is still small. Finally, on the cluster with only a few nodes with greater performance, the best scheduling based on the profiling and benchmarking brings the greatest difference.

At the same time, the architecture and demands of the application affect the difference between scheduling approaches too. The heterogeneity of the application's components allows the scheduler to utilise the differences in a hardware for the better performance of a whole system. Generally, with the increasing heterogeneity of the application's components, the performance gain caused by the benchmarking-based scheduler over the worst scheduling and the standard scheduling grows.

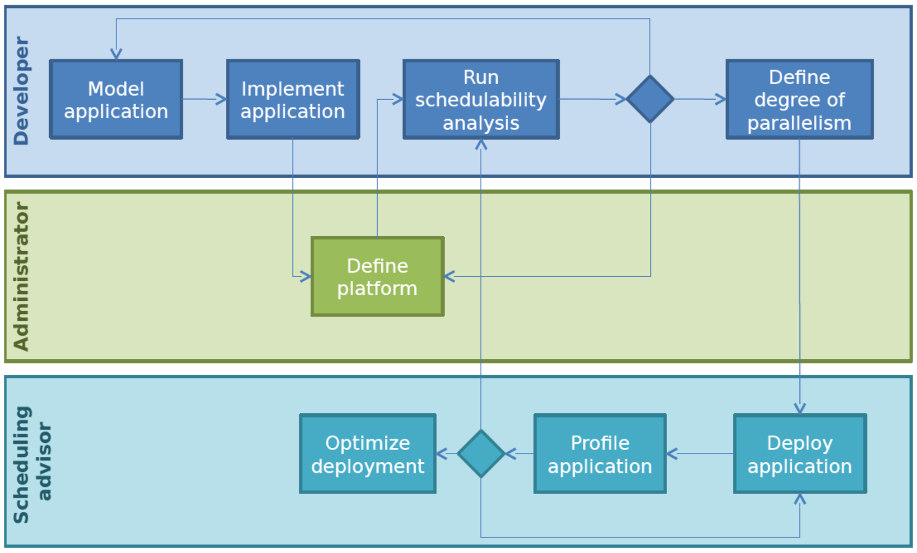
In our experiments, the suitability of the profiling and benchmarking-based scheduling was proven on the application without components that could utilise a special hardware such as GPU or FPGA. Different components of our application only had different demands on the CPU and overall node performance (e.g., the memory speed). The performance gain of our scheduler in the terms of all tuples processed by the whole application was over Storm's standard scheduler 4.1 % and over the 'worst scheduler' 17.5 %. An average gain measured for each component type then was 6.8 % over the standard scheduler and 11.7 % over the 'worst scheduler'. For more detailed statistics from multiple tests on different data see Table 1. The results of our scheduler are satisfying but they still strongly depends on the heterogeneity of a cluster and an application as described earlier. Finally, we assume that the applications employing the GPUs or the FPGAs for some of their components will benefit even more.

Table 1 Scheduler performance comparison – cumulative results of multiple tests based on number of tuples processed in time interval

Component	*W tuples*	*S tuples*	*B tuples*	*S-W gain*	*B-S gain*	*B-W gain*
AnalyserBolt	135,096	163,993	164,714	121.39%	100.44%	121.92%
DownloaderBolt	1,396	1,499	1,494	107.38%	99.67%	107.02%
ExtractFeaturesBolt	39,745	41,867	47,991	105.34%	114.63%	120.75%
FeedReaderBolt	1,580	1,576	1,576	99.75%	100.%	99.75%
FeedUrlSpout	45,711	46,334	45,654	101.36%	98.53%	99.88%
IndexBolt	39,744	41,866	47,989	105.34%	114.63%	120.75%
Total	263,272	297,135	309,418	112.86%	104.13%	117.53%

Notes: W – worst scheduler, S – standard scheduler, B – performance and benchmark-based scheduler, S-W gain – gain of standard scheduler over worst scheduler

Figure 5 The sequence of the individual steps in development and deployment of a JUNIPER application with utilisation of the scheduling advisor (see online version for colours)



7 Utilisation of the scheduling advisor in development process

Outputs of the macro-scheduling component, which were described in the previous section, are produced at run-time, processed by the advisor component, and utilised to improve the assessed JUNIPER applications at their design-time. Development and deployment of a JUNIPER application requires cooperation of different roles of responsible participants who can utilise the advisor's outputs in the development process. These roles are namely:

1 *an analyst*, who describes the required functionality of a JUNIPER application including its time-based constraints which have to be met at run-time

2 *an architect*, who designs and describes the application's architecture in details with respect to the JUNIPER platform

3 *a developer*, who implements the application as a distributed system of the concurrently running components

4 *a system administrator*, who deploys the application to a particular cluster with certain performance characteristics that runs the JUNIPER platform.

The development process of a JUNIPER application and the utilisation of the scheduling advisor in the development and deployment of the application by the above mentioned roles is depicted in Figure 5.

After the *modelling* and *implementation* of the application by an analyst and/or a developer, a system administrator *defines the run-time platform* where the application will be executed. Then, the developer performs the *schedulability analysis* to determine an initial deployment of the application. The application deployment has to meet the computation resources utilisation and a real-time constraints both defined at the design-time; otherwise, if it would not be possible to suitably deploy the application, the application has to be remodelled, reimplemented, and the platform redefined, so another schedulability analysis will result in a suitable deployment. Finally, the developer *defines the degree of parallelism* of the application's components, i.e., the numbers of instances of the individual components) and the application is deployed.

In the next step, the deployed application is executed and profiled by the scheduling advisor concurrently with benchmarking of the run-time platform. The results of the profiling and benchmarking are utilised by the scheduling advisor to optimise the deployment with the current degree of parallelism, as it has been described in Section 5. Moreover, the profiling and benchmarking results can be used by the developer in the repeated schedulability analysis to propose a better initial deployment with a new degree of parallelism.

8 Conclusions

This article described the problems of the adaptive scheduling of the stream processing applications on the heterogeneous clusters and presented an ongoing research towards the novel scheduling advisor. In the article, we outlined general requirements to the scheduling in the 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 the stream processing in heterogeneous clusters, analysed the 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 performing an evaluation of the proposed approach in practice. Our future work mainly aims at possible improvements of the scheduler performance, which is important for the real-time processing, at addressing the problems connected with an automatic scaling of the processing components (i.e., their elasticity), and at addressing the issues related to the eventual decentralisation of the scheduler's implementation.

Acknowledgements

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

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

Aniello, L., Baldoni, R. and Querzoni, L. (2013) 'Adaptive online scheduling in Storm', in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ACM, pp.207–218.

Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J. (2002) 'Models and issues in data stream systems', in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ACM, pp.1–16.

Balazinska, M., Balakrishnan, H. and Stonebraker, M. (2004) 'Load management and high availability in the Medusa distributed stream processing system', in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM, pp.929–930.

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

Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P. and Dongarra, J. (2012) 'DAGuE: a generic distributed DAG engine for high performance computing', *Parallel Computing*, Vol. 38, No. 1, pp.37–51.

Brito, A., Martin, A., Knauth, T., Creutz, S., Becker, D., Weigert, S. and Fetzer, C. (2011) 'Scalable and low-latency data processing with stream mapreduce', in *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, pp.48–58.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y. and Zdonik, S.B. (2003) 'Scalable distributed stream processing', in *CIDR*, Vol. 3, pp.257–268.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K. and Sears, R. (2010). Mapreduce online', in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association.

Dean, J. and Ghemawat, S. (2008) 'MapReduce: simplified data processing on large clusters', *Communications of the ACM*, Vol. 51, No. 1, pp.107–113.

Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D. (2007) 'Dryad: distributed data-parallel programs from sequential building blocks', *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 3, pp.59–72.

Lam, W., Liu, L., Prasad, S., Rajaraman, A., Vacheri, Z. and Doan, A. (2012) 'Muppet: MapReduce-style processing of fast data', *Proceedings of the VLDB Endowment*, Vol. 5, No. 12, pp.1814–1825.

Li, B., Mazur, E., Diao, Y., McGregor, A. and Shenoy, P. (2011) 'A platform for scalable one-pass analytics using MapReduce', in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM, pp.985–996.

Marz, N. (2014) *Apache Storm*, Git Repository [online] https://git-wip-us.apache.org/repos/asf?p=incubatorstorm.git.

Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A. and Hand, S. (2011) '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.

Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010) 'S4: distributed stream computing platform', in *IEEE International Conference on Data Mining Workshops (ICDMW)*, IEEE, pp.170–177.

Vinothina, V., Rajagopal, S. and Ganapathi, P. (2012) 'A survey on resource allocation strategies in cloud computing', *International Journal of Advanced Computer Science and Applications*, Vol. 3, No. 6, pp.97–104.

Warneke, D. and Kao, O. (2011) '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.