This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2907950, IEEE Transactions on Parallel and Distributed Systems

Multi-Level Elasticity for Data Stream Processing

Vania Marangozova-Martin, Noël de Palma and Ahmed El Rheddane
Univ. Grenoble Alpes, CNRS, LIG, F-38000 Grenoble France
E-mail: firstName.secondName@imag.fr

Abstract—This paper investigates reactive elasticity in stream processing environments where the performance goal is to analyze large amounts of data with low latency and minimum resources. Working in the context of Apache Storm, we propose an elastic management strategy which modulates the parallelism degree of applications' components while explicitly addressing the hierarchy of execution containers (virtual machines, processes and threads). We show that provisioning the wrong kind of container may lead to performance degradation and propose a solution that provisions the least expensive container (with minimum resources) to increase performance. We describe our monitoring metrics and show how we take into account the specifics of an execution environment. We provide an experimental evaluation with real-world applications which validates the applicability of our approach.

Index Terms—stream processing, multi-level elasticity, Apache Storm

1 Introduction

Big data is a challenge in various computing system domains. It is present in IoT with the proliferation of connected devices, grows with the increasing scale of high performance computing systems and is coupled with the expanding Internet and social network activities. It is a major topic in the data intelligence business.

There are two major techniques to process big data: *batch processing* and *stream processing*. In *batch processing*, data is first stored in huge databases and is processed later, usually with scalable programming models such as Google's MapReduce [1]. However, with the ever growing size of data, the cost of data transfer and storage becomes prohibitive [2], [3]. Moreover, in multiple domains, what is important is not to keep the initial data but to analyze it as fast as possible to produce valuable intelligence [4], [5]. To tackle these issues, *stream processing* systems put the emphasis on reactivity and analyze data as it is produced. Recent years have seen the emergence of multiple stream processing solutions [6], [7], [8], [9].

If with batch processing the size of data is fixed and known in advance, in stream processing data arrives continuously and at varying rates. Indeed, consider an application detecting *Denial-of-Service* attacks. To reliably detect an attack, the application is to intercept and analyze, on the fly, all incoming service requests. However, the number of requests received within a time period will depend on the clients' activity. There will be calm periods with few requests, as well as peak periods with numerous requests. As the application needs to provide a timely answer in all cases, its resources requirements will vary with the scale of the input workload. It will require few computational resources during calm periods and will increase its demands during active periods. If resource provisioning is statically estimated using a worst case scenario, resources will be underutilized. If the estimation uses an average load, the application will suffer from performance degradation.

The performance problems of static provisioning have motivated the use of *elasticity* where resources are allocated on-demand. If existing proposals share the same goal of minimizing resource consumption while maintaining application performance, they differ in their responses to *why*, *when* and *how* to scale resources. In terms of performance goals (the *why* question), existing solutions pursue cost efficiency [10], [11] or aim at enforcing QoS guarantees on applications' latency [12], [13] and throughput [14]. To decide *when* to adapt the quantity of allocated resources, solutions either try to prevent performance problems through predictive application models [13], [15], or react to performance degradation using application monitoring. As for the *how* question, solutions use migration to relieve overloaded resources [16] or replication to adapt the parallelism degree of computations [12], [13], [14], [17]. In all cases, however, the elasticity strategy is elaborated with a focus on the application and abstracts many issues related to the execution platform. In particular, existing proposals have a homogeneous view of execution resources and ignore the hierarchy of execution containers (VMs, processes, threads) onto which application components are mapped. In this work, a *container* is any entity that provides execution resources to a computation.

In our proposal we tackle the elasticity problem the other way round: we focus on the execution environment and investigate how changing the number of different execution containers impacts application performance. The basic idea is that different execution containers have different capacities that come with different costs. Provisioning a higher capacity container, typically a VM, is more costly, in resources and time, than provisioning a lower-capacity one, such as a process or a thread. However, a thread only exists in the context of a machine and cannot be provisioned of there are no available resources. Our goal is to automatically provision the least expensive resources capable of satisfying the applications' performance needs. In a hierarchy of containers, our solution leverages the resources of higher-capacity containers using lower-capacity container consolidation.

Considering the widely spread pay-as-you-go cloud computing model, we propose a container-aware reactive elasticity solution for streaming applications. We place ourselves in the context of Apache Storm [7] and investigate the different conditions requiring the provisioning of different execution containers. We identify when a streaming application can scale using lower-level execution containers (and stay within the limit of already provisioned VMs) and when new VMs should be provisioned.

More in detail, our contributions are:

- **Metrics for Application Congestion.** We propose a simple metric for detecting local congestion in streaming applications. The idea is that an application component becomes a bottleneck if it cannot absorb all incoming data i.e. if its data processing rate is lower than its arrival rate.
- **Multi-level elastic resource provisioning.** We propose an elastic strategy which considers different levels of execution containers. The intuitive idea is to provision the minimum number of *heavy* containers (e.g. VMs or processes) and to maximize resource usage by multiplying the number of *lightweight* containers (threads). More importantly, the strategy accounts for the performance impact of different execution containers as provisioning the wrong type of resource can actually decrease performance. We devise a simple yet efficient strategy to benchmark applications and to experimentally discover the minimal (cheapest) configuration for a given workload.
- **Transparent integration into Apache Storm.** We implement our proposal in Apache Storm [7], a top-level Apache project supported by an active community including large companies such as Twitter and Yahoo. Our implementation preserves the Storm API and provides for provisioning of execution containers at different levels. We automatically benchmark applications, discover the minimal configurations to support a given workload and generate elasticity controllers to manage application bottlenecks. We validate our approach with two real-world applications cases. The first is an industrial DDoS[1] application processing real workloads. The second is an online data analysis of results produced by the CoMD [18] simulation application.

The paper is organized as follows. Section 2 investigates the performance impact of different execution containers on stream processing applications. Section 3 presents our elasticity proposal. Section 4 discusses the implementation in the Storm processing environment. Section 5 presents the evaluation of our approach in the cases of the DDoS and CoMD analysis applications. We discuss related work in Section 6 and conclude in Section 7.

2 Motivation for Multi-Level Elasticity

After introducing the system model (Section 2.1) and the performance metrics (Section 2.2) we consider, we present our study of Apache Storm (Section 2.3). We show that provisioning execution containers at different levels has different performance impacts and that non considering the container hierarchy may lead to performance degradation.

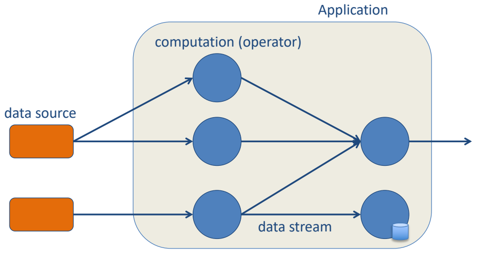


Fig. 1: A stream processing application

2.1 System Model

Our application model captures the common features of most existing streaming processing environments [6], [7], [19], [20], [21]. In this model, an application is represented by a directed acyclic graph whose nodes represent computation operators and edges represent data streams (Fig. 1). The operators are application-specific and may range from simple filters to complex data transformations. The data streams are composed of *tuples* each containing a key and a payload. The payload is the data to be processed while the key is used for grouping and routing data streams among operators. The input of an application is data produced by external systems. Among others, it may ship logging information, customer tracking data and sensor measurements. The output may be a visual representation, a result saved on a persistent storage or a data stream.

An application is deployed on a cluster of execution nodes which are managed by a master node (Fig. 2). The master monitors the nodes and manages scaling and recovery. The nodes provide hierarchically organized execution containers that serve as execution support for applications. Typically, a node may be a virtual machine running multiple multi-threaded processes.

Application computations may be parallelized using multiple instances. These instances are mapped to the smallest containers i.e the ones at the bottom of the container hierarchy. Changing the number of instances requires a change in the level of parallelism of execution containers which
are ex
be cre
proces

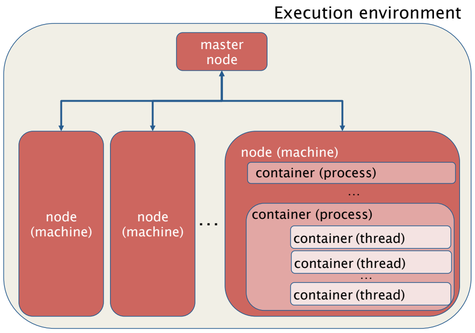


Fig. 2: Stream processing execution environment

[1] Detection of Denial-of-Service

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2907950, IEEE Transactions on Parallel and Distributed Systems

2.2 Performance Metrics

In the context of stream processing, the major performance criterion is the application's capacity to process the input workload and produce a timely result. We do not consider the application operators in the initial topology graph but characterize the processing activity of individual instances. We propose to use the following sampling metrics:

- *Number of received tuples.* This metric, denoted by $received_T(c)$, quantifies the tuples received by an instance c for a given period of time T.
- *Number of processed tuples.* This metric, denoted by $processed_T(c)$, quantifies the tuples processed by an instance c for a given period of time T.
- *Processing activity.* Using the previous metrics, we define the $health$ of an instance as the proportion of the tuples it processes and the tuples it receives for a given period of time T:

  $$health_T(c) = \frac{processed_T(c)}{received_T(c)}$$

  If an instance has a $health$ of 100% it is able to timely process all incoming tuples. On the contrary, if $health_T(c) < 100\%$, there are tuples waiting to be processed. If this situation lasts, the instance becomes a performance bottleneck for the application.

Whether an application meets its specific time constraints depends on multiple factors including the available resources on the used execution nodes, the properties of the network connections, the variations in the input workload and the application computations. In this work we consider typical streaming applications that process a high number of small-size messages (e.g. Twitter posts, sensor log messages, etc.). The performances of such applications are bound by the available computation resources and do not suffer from network saturation. As the resources provisioned for the applications are the ones allocated to the application's execution containers, at the level of an execution node they are limited by the node's memory and CPU capacity. We therefore monitor resource usage using the following metrics:

- *CPU occupation.* This metric, denoted by $totalCPU_T(n)$, provides the average CPU usage of a given execution node n for a given period of time T.
- *Memory occupation.* This metric, denoted by $totalMem_T(n)$, indicates the average memory usage of a given node n for a given period of time T. Like the previous metric, it is given as percentage of the total available memory.

Given these two metrics we can follow the execution load of a node. If neither is saturated, the node has unused resources that can be allocated to additional containers. If either CPU or memory usage goes over a threshold, the node is considered saturated and needs an appropriate scaling decision.

2.3 Performances of Storm

To illustrate the fact that the level of parallelism of different execution containers affects differently application performance, we have used Apache Storm [7]. Storm is a top-level Apache project providing a fault-tolerant distributed stream processing framework. Storm applications are called *topologies* and are composed of *spouts*, which act as data stream sources, and of *bolts*, which process (receive, transform and emit) data streams.

The execution environment of Storm distinguishes four levels of execution entities, namely nodes, workers, executors and tasks. The nodes are the physical or virtual machines on which the application runs. *Workers* are processes, i.e. Java virtual machines, whose number per node is configured by the administrator. Workers contain *executors*, i.e. Java threads, that execute *tasks* that correspond to the instances of spouts and bolts.

We have used Storm 1.2.2 on a virtualized cluster managed by OpenStack [22]. The physical nodes are provided by the Grid'5000 platform [23] and feature 2xIntel Xeon E5-2630 v3 CPUs with 128 GB of RAM, 2x600 GB HDD and 2x10 Gbps network. The OpenStack stack has been itself deployed using the *enos* tool [24]. For the virtualized cluster we have used three classical VM flavors: m1.small (1 vCPU, 2GB RAM, 20 GB disk), m1.medium (2 vCPU, 4GB RAM, 40 GB disk) and m1.large (4 vCPU, 8GB RAM, 80 GB disk). The deployment and measurement process for all experiments has been scripted and is automatic and reproducible.

In the following we show the results from a simple filter application deployed on m1.large nodes. The application consists of one spout sending strings to one bolt that matches them against known patterns. We make the spout emit a linearly increasing load and consider the processing capacity of the bolt. We deploy the bolt on one node and vary the number of tasks, executors and workers. For this 4vCPU VM configuration, we have made tests with 1, 2 and 4 workers, 1, 2, 4 and 8 executors per worker and 1, 16 and 64 tasks. All experiences have been executed 5 times. The graphics illustrate representative results and show the mean values with the standard deviation error.

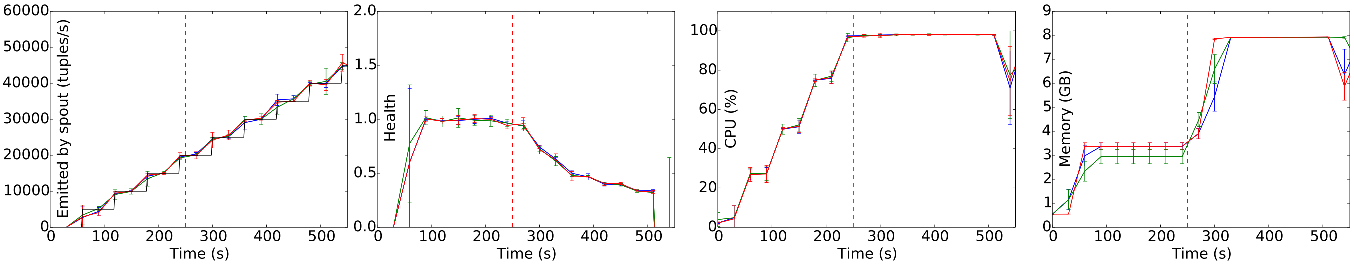
2.3.1 *Task parallelism*

When varying the number of tasks, we observe the same system behavior in all considered configurations. In Fig. 3 we show the results for the configurations with one worker and four executors. 1-4-4 instantiates one task per executor, 1-4-16 instantiates 16 tasks (4 tasks per executor) and 1-4-64 has 64 tasks (16 tasks per executor). The three graphics being very similar, the important conclusion to draw from these results is that increasing task parallelism has no impact on the application performance.

In fact, tasks have a different nature compared to workers and executors. If the latter are clearly related to resource allocation, tasks are logical units of computation and a single task is executed by a single executor. In a Storm topology, the number of tasks actually defines the maximum number of executors this topology can use. As a consequence, the tasks' number should be big enough to not limit performance (cf. Section 2.3.2).

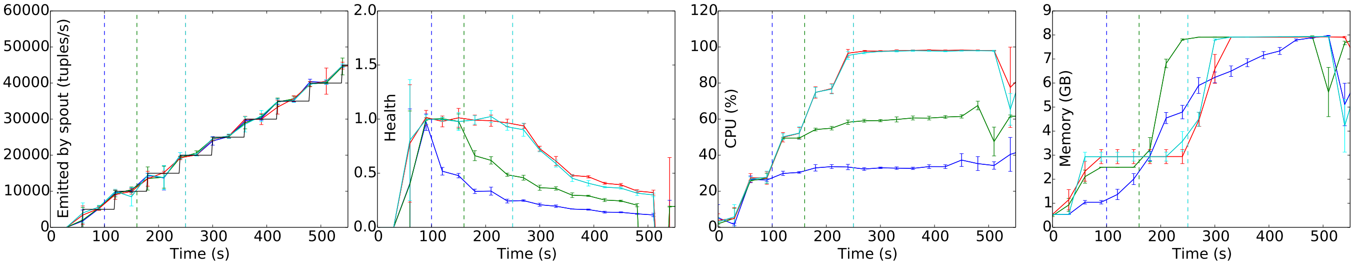
If we look at the performance results more in detail, we gain some interesting insight into the system's dynamics. First, in Fig. 3a we see that the spout succeeds in injecting all the input workload in the system as the colored lines of the spout's emitted tuples closely follow the black line of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2907950, IEEE Transactions on Parallel and Distributed Systems



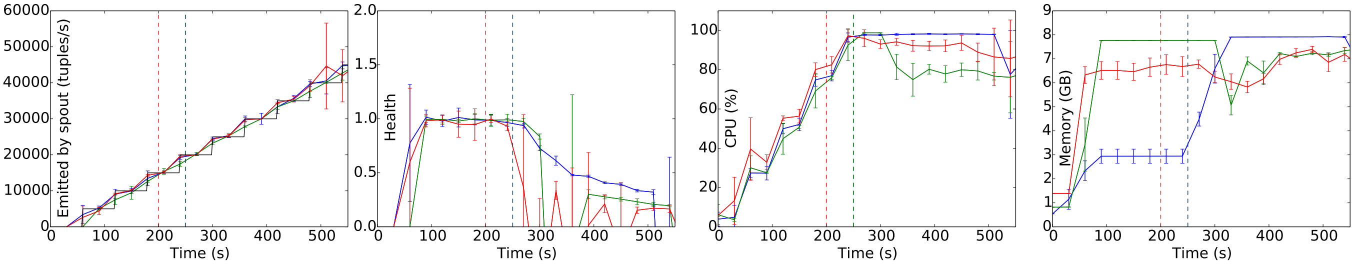
(a) Input workload (b) Bolt processing activity (c) CPU usage (d) Memory consumption

Fig. 3: Impact of task parallelism. Plotted configurations (workers-executors-tasks) are 1-4-4, 1-4-16, 1-4-64.



(a) Input workload (b) Bolt processing activity (c) CPU usage (d) Memory consumption

Fig. 4: Impact of executor parallelism. Plotted configurations (1 worker-x executors-1 task per executor) are 1-1-1, 1-2-2, 1-4-4, 1-8-8.



(a) Input workload (b) Bolt processing activity (c) CPU usage (d) Memory consumption

Fig. 5: Impact of worker parallelism. Plotted configurations (workers-executors-tasks) are 1-4-4, 2-4-4, 4-4-4.

workload. Then, if we look at the bolt's activity in Fig. 3b, we observe that the bolt successfully processes all incoming tuples until around $250s$ and then its health declines. Given the values of the workload at $250s$, we deduce that the bolt processes successfully up till 20000 tuples per second ($tuples/s$).

The explanation of the bolt's degraded performance after $250s$ can be found in the monitoring information about the CPU and memory usage of the execution node. Fig. 3c shows that at $250s$, the CPU usage of the node reaches 100%. As a result, the bolt's computational resources reach a limit and it cannot further increase its processing capacity. As incoming tuples need to wait, they are stored in memory and provoke an increased memory consumption (Fig. 3d). When the memory is saturated, the node crashes.

2.3.2 Executor parallelism

The level of parallelism at the executor level increases the bolt's performance as long as the number of executors does not exceed the number of CPU cores (Fig. 4). Let us consider the configurations with one worker, one task per executor and one (1-1-1), two (1-2-2), four (1-4-4) and eight executors (1-8-8). We see that passing from one to two and then to four executors (one to two to four cores in our 4vCPU VMs) improves the bolt's performance (Fig. 4b), while passing from four to eight does not (red and cyan configurations). With 1-1-1, the bolt's health goes below 0.9 at $100s$. With 1-2-2, this happens at $160s$. For the last two, their health declines at $250s$. Correlating this with the incoming workload (Fig. 4a) gives that compared to the 20000 $tuples/s$ of the red and cyan configurations, the blue and green ones respectively reach only 5000 $tuples/s$ and 10000 $tuples/s$. Here again, the performance is directly related to resource usage. When the CPU saturates (Fig. 4c), the load cannot be processed in time and the bolt's health declines. The memory consumption starts increasing (Fig. 4d) and ultimately leads to a crash.

2.3.3 Worker parallelism

Provisioning more than one worker decreases the performance. If this is obvious in the case of one-vCPU machines where multiple processes compete for the CPU, the effect is more subtle in the case of multiple processors. If we consider three configurations featuring respectively one (1-4-4), two (2-4-4) and four (4-4-4) workers on our 4vCPU nodes (Fig. 5b), we see that the red configuration is the first to decline and crash. For the two others we do not observe any difference while the bolt is in good health but looking further we see the two-worker configuration is less stable and crashes sooner. The explanation lies in the fact that multiple workers occupy more memory and reach more easily the node's limit (Fig. 5d).

2.3.4 Conclusions

Our experimental results show that adapting the level of parallelism of the different Storm containers has different impact on the processing performance. If tasks do not really influence performance, increasing the number of executors, up to the limit of physically available resources, does increase performance. On the contrary, increasing the number of workers leads to performance degradation.

3 A Strategy for Multi-Level Elasticity

We consider the system model presented in Section 2.1 in which an application is executed on a cluster of nodes. The environment is not multi-tenant and the nodes are dedicated to the application.

An application is deployed in containers which are hierarchically organized. Each instance is supported by a dedicated container at the finest granularity (lowest) level. To be provisioned, such a container needs to be encapsulated in a full container hierarchy. This typically translates the fact that a thread cannot exist outside a process which needs a machine to be executed. Lower-level containers (e.g. threads) need less computational resources than higher-level ones (e.g. processes) and are therefore less costly and faster to provision.

We suppose that we are in the case of streaming applications processing small-size messages and therefore that network connections are wide enough. Possible congestion is therefore caused by insufficient computational resources (memory and CPU) allocated to the execution containers of application computations.

If an application operator is parallelized, the data stream it needs to process is evenly partitioned among its instances.

The stream processing system is able to manage the state of stateful application operators as in [7].

3.1 Elasticity Control

We propose a reactive approach which monitors the computation activity of an application using the metrics introduced in Section 2.2. Basically, if the application does not process the input workload fast enough, it needs a *scale out* and is to be allocated more resources. If it can process its workload with less resources, it is to be *scaled in*.

We build automatic elasticity controllers that take into account the specificity of both the target execution environment and the target application. We propose a three-step methodology which considers in order: 1) the performance impact of execution containers, 2) the discovery of appropriate application configurations for a given input workload and 3) the construction of application-specific elasticity controllers.

The first step deals with the container hierarchy in the system model and the fact that different types of containers may be provisioned to provide the resources needed by an application. To be able to decide what containers to provision, we need to know how the parallelism level of different containers impacts performance. This information may be already available after an extensive use of the platform, be provided by a platform administrator or be discovered through platform benchmarking, as in our Section 2.3.

In the second step, we automatically benchmark the application to establish a relation between a given level of input workload and a suitable application configuration. We start with a minimal application configuration in which all application operators are not parallelized and run on the same machine. The idea is to vary the level of input workload and to discover the needed parallelism degree (and the corresponding containers) for application operators.

Algorithm 1 Application benchmarking

```
 1: procedure BENCHMARKAPPLICATION
 2:     for $inputLoad \leftarrow 0$ to $initLoad$ step$= \Delta L$ do
 3:         $injectLoad(inputLoad, \Delta)$
 4:     end for
 5:     for $inputLoad \leftarrow initLoad$ to $maxLoad$ step$= \Delta L$ do
 6:         $injectLoad(inputLoad, \Delta)$
 7:         while $in\ current\ \Delta\ time\ interval$ do
 8:             find $i \in sortedInstances$ such that
 9:                 $health(i) < minHealth$
10:             if $i \neq null$ then
11:                 SCALEOUT($i.getOperator(), inputLoad$)
12:             end if
13:         end while
14:     end for
15: end procedure
```

The benchmarking process is described in Algorithm 1. It works with a linear load injector sending a given number of tuples per second ($tuples/s$) to the application. We consider an initial system load ($initLoad$) and a maximum system load ($maxLoad$). The application is benchmarked for the values in the interval using a load incrementation step (ΔL).

If the initial load to consider is greater that 0 $tuples/s$, the system is to be warmed up until $initLoad$ is reached (lines 2-4). Then, each value for the system load is injected during a fixed time interval Δ (lines 5-6). During this time interval, the application is periodically monitored to detect congestion. If an application is composed of multiple parallelized operators, the monitoring considers each computation instance individually. The order in which the instances are considered is given by their distance to the application data sources (lines 8-9). We prioritize the instances that are closer to the source of information following the logic that if there is bottleneck upstream, it will affect the performance of instances that are further downstream.

An instance is considered to undergo performance problems when the number of processed tuples falls behind the number of received ones. When the ratio (the instance's $health$ metric) goes below a given threshold ($minHealth$), we consider the instance to be congested and trigger a scale-out which increases the parallelism degree of the corresponding operator (line 11).

The scale-out at the operator level (Algorithm 2) starts by searching for the least loaded node with sufficient resources (line 2). As the processing capacity of an application opera-

Algorithm 2 Operator scale out

```
 1: $List\langle Node\rangle\ nodes$;
 2: $Map\langle Operator, Configuration\rangle\ confs$;
 3: procedure SCALEOUT(Operator op, Load inputLoad)
 4:     find $node \in nodes$ such that
 5:         $totalCPU(node) = MIN_{n \in nodes}(totalCPU(n))$
 6:     if $isOverloaded(node)$ then
 7:         $node \leftarrow deployNode()$
 8:         $nodes.update(node)$
 9:         $container \leftarrow deployContainerHierarchy(node)$
10:     else
11:         $container \leftarrow deployOptimalContainer(retNode)$
12:     end if
13:     $instance \leftarrow op.createInstance()$
14:     $op.updateInstances(instance)$
15:     $confs.update(inputLoad, op, instance, container, node))$
16:     $deploy(instance, container)$
17: end procedure
```

tor is mainly related to the CPU, we use the totalCPU metric (Section 2.2). If all CPUs are fully exploited or the memory usage is beyond a given threshold (line 3), we consider that the node cannot efficiently support the execution of a new operator instance. In this case, a new machine is to be provisioned and a full container hierarchy is to be deployed (lines 4 and 6). In our experiments a simple approach with a threshold of 0.9 for both CPU and memory has been sufficient. In other words, we have considered a node to be overloaded if $totalCPU > 0.9$ *or* $totalMem > 0.9$.

If the node still has available resources, the new instance is to be deployed in an optimal configuration providing the least expensive container to increase performance. This decision depends on the platform-specific information we have obtained in the first step and is encapsulated in the $deployOptimalContainer$ operation (line 8). In the case of the Storm platform from the previous section, this operation will typically capture the knowledge that on an existing node, a new instance should be created in the already existing worker but should be supported by a new executor with one task.

The algorithm keeps in memory the fact that the operator *op* has scaled-out when its input workload has reached $inputLevel$ and that the corresponding execution configuration now contains a new $instance$, executing in the $container$ on the $node$ node (line 12). We consider that the time period Δ is long enough to allow for application stabilization. In other words, if there is a need to provision more instances for the scaled-out operator or to increase the parallelism degree of other instances downstream, all scaling iterations are managed within Δ. Thus, at the end of this period, the system has discovered and saved an application configuration which is capable of processing the workload at a level $inputLevel$ without congestion. The configuration has been constructed incrementally by provisioning the least expensive resources at each step and ensuring that provisioned nodes are used at maximum.

The information about the configurations discovered during the benchmarking step is used in the final third step which creates a dedicated elasticity controller. Knowing the levels of input workload that are successfully processed by the different configurations, it is possible to deduce their workload thresholds. The lower threshold of a configuration indicates that for lesser workloads the application may use smaller configurations. The higher threshold gives the maximum workload level this configuration can support. Using the corresponding map, the elasticity controller monitors the variations of the input traffic and decides whether to trigger a scale-in or a scale-out. If the monitored level of the input workload is within the thresholds of the current execution configuration, the controller does nothing. If the input traffic exceeds the upper threshold of the current configuration, the controller computes the needed containers to reach the configuration that supports the new level of workload and triggers a scale out. Symmetrically, if the traffic goes below the lower threshold of the current configuration, the controller triggers a scale-in to reach a reduced execution configuration. The victim nodes and executors are chosen in a LIFO order: the last containers to be provisioned are the first to be freed. The rationale behind this strategy is to be able to free nodes (VMs) as soon as possible.

To prevent oscillation, instead of possibly scaling at each observation, the variation trend may be computed with linear regression over several observations. In the case of Storm, our experience has shown that monitoring with the default frequency of $1minute$ yields stable observations. Monitoring every $10seconds$ however provides results with significant variations where a scaling decision would need at least three successive observations.

4 Implementation

To integrate elasticity control in Storm, we substitute the default Storm scheduler with a scheduler interacting with an elasticity controller (cf. Fig. 6). Both the scheduler and the controller run on the Storm's *Nimbus* management node and therefore have access to both the cluster resources and the application configuration.

Our scheduler is seamlessly integrated in Storm as it implements the $IScheduler$ interface. It replaces the Storm's default round-robin strategy with a strategy that takes into account reconfiguration orders issued by the elasticity controller. Following the strategy presented in the previous section (Section 3), the controller triggers reconfigurations as reactions to applications' monitoring information. It *scales out* an application operator when it detects a bottleneck on one of its instances. It *scales in* the application by going back to a configuration supporting the decreased traffic level.

During scaling operations, we minimize the overhead as the scheduler does not rebalance the whole application but maintains the existing mapping between computation instances and Storm's execution containers. During a scale out, all existing instances continue to run where they have been deployed. Only new instances are started on the newly provisioned resources. During a scale in, the scheduler affects only the instances to be destroyed.

The elasticity controller puts together resource management, application monitoring and application scaling. For resource management, we consider the deployment of Storm on a cloud platform and use the available resource provisioning features. We have defined an unified cloud interface and have implemented it for Azure, Amazon,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2907950, IEEE Transactions on Parallel and Distributed Systems

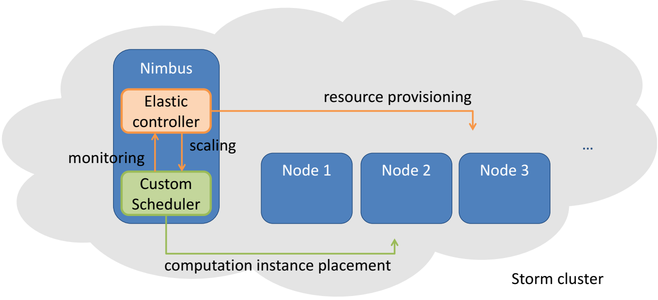


Fig. 6: Elastic Storm architecture

OpenStack and VMWare clouds. The controller uses this interface to provision or free Storm nodes and thus has an up-to-date information about the computation nodes (virtual machines). When provisioning new nodes, the controller configures them with the appropriate number of workers (one) and inserts them into the Storm cluster.

For application monitoring, the controller uses the scheduler's Thrift interface [25], [26]. To compute the application-related metrics presented in Section 2.2 we use the information about the application topology, as well as about $emitted$ and $executed$ tuples. To compute the number of received tuples by an operator instance for a given period of time, we consider the sum of the emitted tuples by its predecessors. The $health$ metric which characterizes the processing activity of a given instance thus becomes

$$health_T(i) = \frac{executed_T(i)}{\Sigma_{predecessor} emmited_T(predecessor, i)}$$

To have access to the metrics characterizing the machine load, we use the metrics on CPU and memory consumption provided by the Ganglia monitoring system [27].

5 Experimental Evaluation

In this section, we show the validity of our work with two real-world applications and two elasticity contexts. We consider an application for detection of denial of service attacks (DDoS) and an application for online data analysis of HPC simulation results (CoMD). We show how reactive elasticity may be used for application benchmarking to discover applications resource needs and how this information may be used for proactive elasticity.

The experimental setup is the one given in Section 2.3 and summarized in the following table.

Hardware	2xIntel Xeon E5-2630 v3 CPU, 128GB RAM, 2x600 GB HDD, 2x10 Gbps
VM mgt.	OpenStack stable/queens
DDoS VM	m1.medium (2vCPU, 4GB RAM, 40GB HDD)
CoMD VM	m1.large (4vCPU, 8GB RAM, 80GB HDD)
OS	Ubuntu 18.04
Storm	Version 1.2.2

The section tackles the DDoS and CoMD application in order.

5.1 The DDoS application

Distributed Denial of Service (DDoS) attacks are a major threat and their detection and prevention is a challenge for service providers. They consist in sending a great number of requests to a network target in order to overload it.

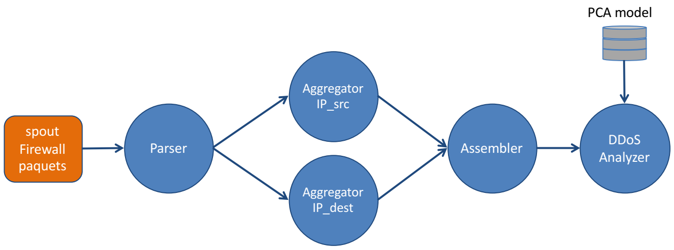


Fig. 7: The DDoS topology

In our work we use a DDoS detection application that initially has been proposed in [28] for the needs of *eolas* green data centers [29]. As in many detection systems, the DDoS application distinguishes between normal and abnormal network traffic. It uses the PCA statistical method [30] to continuously learn about the changing system behavior and is able to detect even *never seen before* attacks.

The application is structured as a lambda architecture [31] including both a batch system and a real-time processing system. The batch system is used to analyze the datacenter traffic and generate the PCA models for normal and abnormal behaviors. The stream processing system is used to analyze the incoming traffic in real-time against the computed behavior models and detect attacks.

In our elasticity experiments we focus on the real-time processing part of the application which is implemented using Apache Storm. This part (Fig. 7) consists of several components. It starts with a *spout* that receives the packets from the datacenter firewalls and injects the corresponding tuples into the topology. A *parser* bolt groups the tuples by source and destination IP addresses and computes statistics about the packets' inter-arrival time. *Aggregator* bolts periodically compute the number and the total payload of the tuples received for the last time period as defined by the PCA model. An *assembler* regroups the tuples produced by the aggregators to match the initial $(IPsource, IPdestination)$ couples. Finally, at the end of each period, an *analyzer* receives a single tuple of aggregated traffic, applies the PCA model and produces a list of identified attacks.

The analysis of real traffic logs shows that the workload of the DDoS detection application is time-periodic. As shown in Fig. 8, working days, as well as week-ends, have similar traffic profiles.

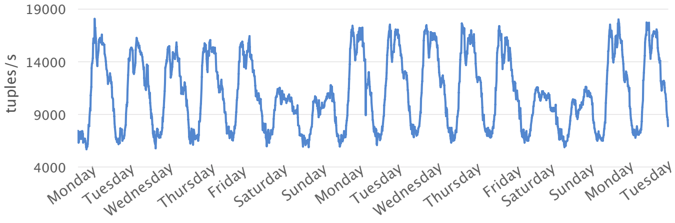
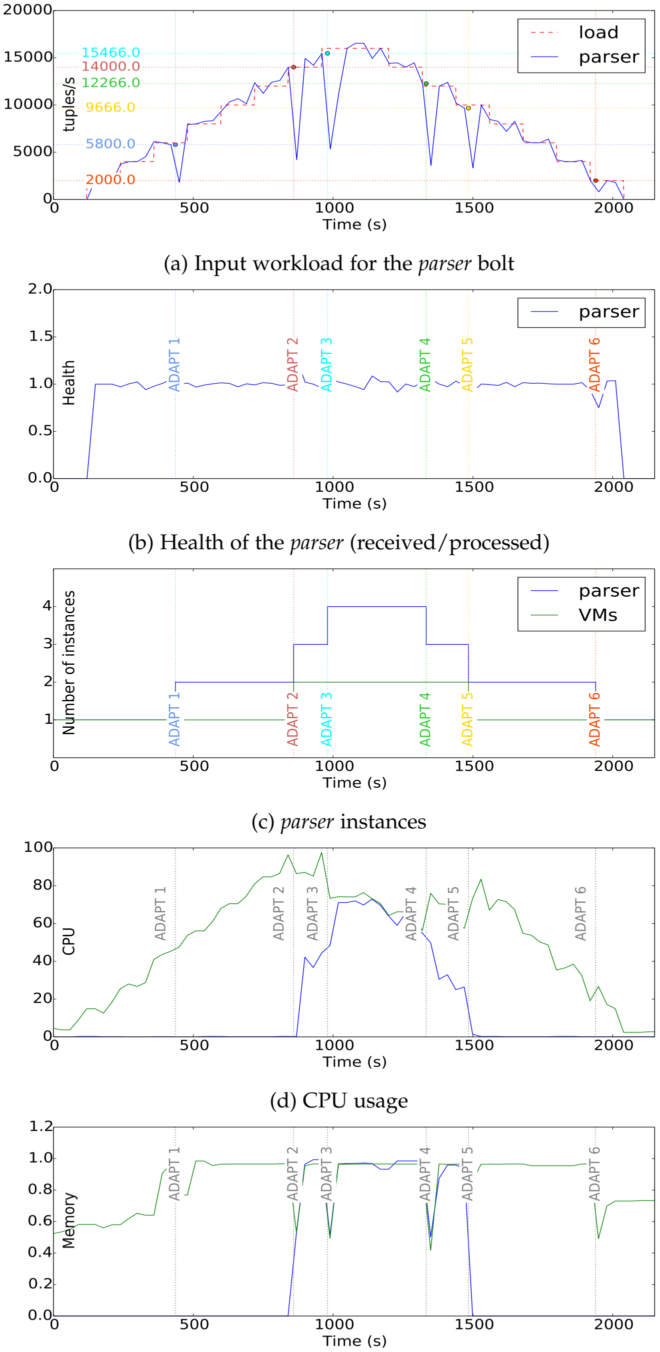


Fig. 8: The DDoS periodic traffic at eolas

5.2 Reactive elasticity for application benchmarking

In this section we show how we use our reactive elasticity approach (Section 3) to benchmark the DDoS application.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2907950, IEEE Transactions on Parallel and Distributed Systems



(a) Input workload for the *parser* bolt

(b) Health of the *parser* (received/processed)

(c) *parser* instances

(d) CPU usage

(e) Memory usage

Fig. 9: Reactive elasticity for DDoS

We use the traffic logs that give the limit (lowest and highest) values of the typical application workload, detect when the application suffers from bottlenecks and use elasticity to provide a resource-sufficient configurations.

We start with a minimal execution configuration. We use one execution node (one virtual machine) on which run all DDoS bolts. The node runs one worker and each bolt is initially supported by one executor with one task.

The workload we inject in the DDoS application starts at 0 $tuples/s$, increases linearly to 16000 $tuples/s$ and then decreases back to 0 $tuples/s$. To change the workload level we use a step of 2000 $tuples/s$. Each workload level is maintained during 2 $minutes$ and the application is monitored at a 30 $seconds$ frequency.

Following Algorithm 1 presented in Section 3.1, the system automatically discovers that increasing the input workload of the DDoS application causes bottlenecks but only at the *parser* level. The other bolts do not suffer from bottlenecks as they receive less traffic or execute simpler computations. The *aggregators* receive half of the tuple flow and perform simpler computations. As for the *assembler* and the *analyzer*, their incoming traffic is negligible as they receive only one tuple per PCA period (1 $tuple/minute$).

As providing more resources to the *parser* solves the bottleneck and does not shift the problem further down the topology, in the following we focus on the elastic management of the *parser* bolt.

The observations related to the application of our reactive elasticity management are shown in Fig. 9. In the first part (Fig. 9a), we show a zoomed version of the input workload of the *parser* bolt. We see that it succeeds in following the load injected by the spout.

The colored lines (`ADAPT1`, `ADAPT2`, etc.) indicate the moments when elastic reconfigurations are triggered. When increasing the input workload, they indicate the moments when the *parser*'s health drops under the decided minimum threshold. In our experiments, the threshold is 0.9 meaning that we trigger elastic reconfigurations when the *parser* fails to process more than 10% of its input tuples. When decreasing the volume of the input traffic, the colored lines (`ADAPT4`, 5 and 6) correspond to reconfiguring to smaller resource configurations. Fig. 9b shows that our elastic reconfigurations succeed in maintaining the *parser*'s health and ensure the processing of the incoming tuples.

Fig. 9c shows the evolution of the provisioned resources for the application. We see that the initial configuration, with one instance of the *parser* (one executor in one worker on one VM) succeeds in treating the incoming tuples for the period between 0 and 480s. Correlating this with the input traffic shows that in this configuration, the *parser* is able to process up till 4000 $tuples/s$ but is not able to support the next workload level of 6000 $tuples/s$. To be able to process more, the *parser* is elastically provisioned with more resources. We increase the parallelism degree of the *parser* and create a second instance. Given that neither the CPU (Fig. 9d), nor the memory (Fig. 9e) are fully exploited on the already provisioned VM, this second instance is provisioned on the same machine but with a different executor. The configuration thus becomes one VM with two executors (Fig. 9c). This configuration succeeds in restoring the health of the *parser* and maintains it until `ADAPT2`. The two *parser* instances are able to timely process a workload going from 4000 $tuples/s$ to 12000 $tuples/s$. Beyond this point, as the CPU of the VM is fully occupied (it is a 2vCPU VM), to create a third *parser* instance we need to provision a new virtual machine. After `ADAPT2`, the execution configuration contains therefore two virtual machines and three executors (Fig. 9c). This configuration supports the traffic of 14000 $tuples/s$ but to go to 16000 $tuples/s$ it needs yet another *parser* instance (`ADAPT3`).

When the input workload decreases, the application is reconfigured to shrink to the configurations discovered to support the corresponding load. The correspondence between execution configurations of the parser and the supported workload thus becomes the one given in Table 1.

Workload level (tuples/s)	Configuration (Σ(VMs,executors))
0 - 4000	(1,1)
4000 - 12000	(1,2)
12000 - 14000	(1,2)+(1,1)
14000 - 16000	(1,2)+(1,2)

TABLE 1: Benchmarking results for the DDoS application

Let us compare our solution to a standard cloud elasticity solution where the resource provisioning is not hierarchical but done at the VM level. With the latter, the ADAPT1 and ADAPT3 reconfigurations would trigger the provisioning of two virtual machines instead of two lower-level execution contaners (threads). Deploying four m1.medium (2-vCPU) VMs instead of the two in our solution would not only be slower but also resource inefficient as the VMs' CPU usage would be of 50%.

5.3 Proactive elasticity for DDoS

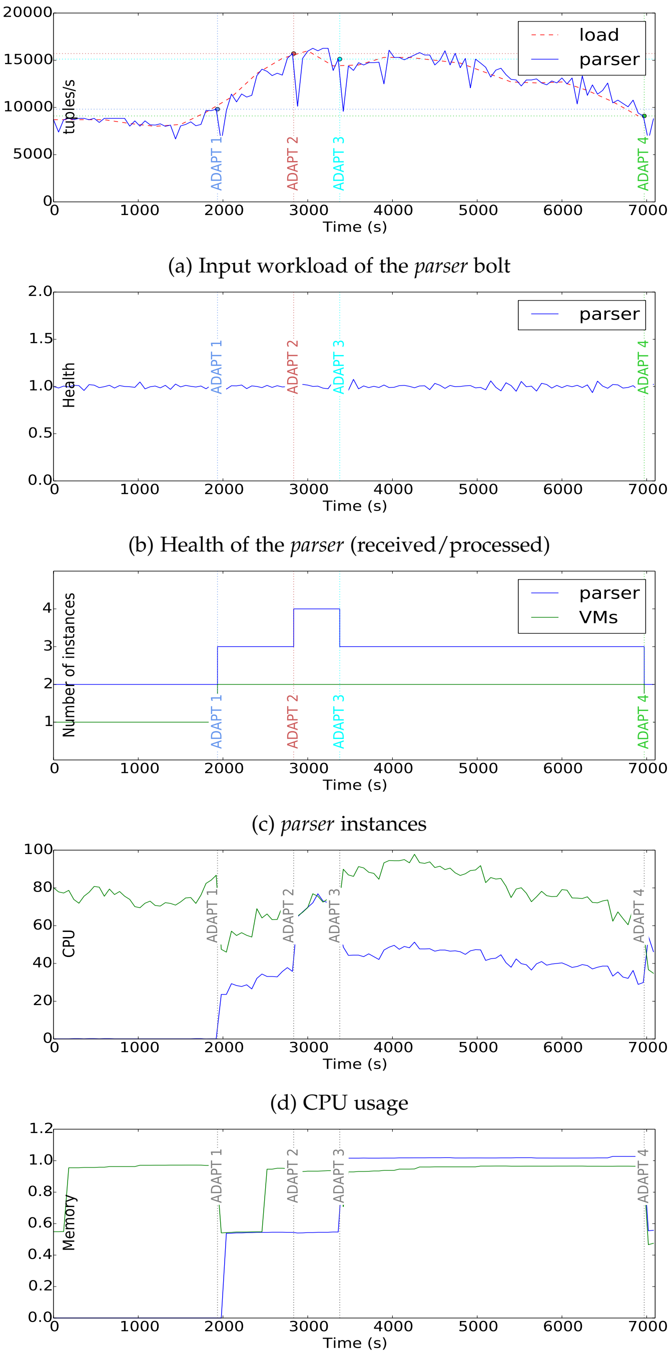
Given that we have information on the periodic nature of the input traffic of the DDoS applications, we can trigger the elastic provisioning of the required resources proactively (Fig. 10). The first part of the figure (Figure 10a) shows the input workload of the *parser*. It has been generated to reproduce the working day traffic of the DDoS application using the mean values per hour. In our experimentation, each such value is maintained 5 minutes.

The traffic variations (Figure 10a) are met with the corresponding parser configurations (Figure 10c). Following the benchmarking results of Table 1, the parser starts using two instances (two executors on one VM) till ADAPT1 as it receives more than 4000 $tuples/s$. At ADAPT1, the controller takes into account the prediction that the traffic will exceed the threshold of 12000 $tuples/s$ and reconfigures the parser. The parser requires the provisioning of a new VM and starts running with three instances. At ADAPT2, the parser goes to four instances on two VMs.

At ADAPT3, the traffic has gone below 16000 $tuples/s$ so the parser is reconfigured to go back to three instances. It goes back to two instances at the end of the day.

The controller triggers application scaling in advance to take into account the time for virtual machine provisioning and application reconfiguration. As in our experimentation the time for the DDoS reconfiguration is insignificant compared to the time for VM provisioning, which in average takes about 1 $minute$, the controller is programmed to order reconfigurations about 2 $minutes$ in advance. This delay is itself negligible compared to the considered traffic period of 1 $hour$ for which the application is reconfigured.

Figures 10d and 10e show that the application saturates the CPU of the first VM before scaling to two VMs. Figure 10b shows that the elastic changes of the parallelism degree of the parser succeed in keeping its health at 1 which means that incoming tuples are processed in time. Our elasticity decisions thus ensure that the DDoS application absorbs its normal input traffic with minimum resources. If the workload goes beyond the expected traffic level, the application can switch to reactive mode but more importantly, trigger an alert for possible DoS attacks.



(a) Input workload of the *parser* bolt

(b) Health of the *parser* (received/processed)

(c) *parser* instances

(d) CPU usage

(e) Memory usage

Fig. 10: Proactive elasticity for DDoS

5.4 Resource dimensioning for online data analysis

Our second use case lies in the context of interconnecting HPC[2] simulations with stream processing applications for online result analysis. We consider the CoMD application which features classical molecular dynamics algorithms [18], [32]. This application simulates the evolution (in terms of energy and position) of atoms using timesteps (iterations). The size of the CoMD simulation is given by three parameters (nx, ny, nz) that define the number of considered atoms (total_number_atoms=4*nx*ny*nz).

[2] High Performance Computing

The default values for these parameters are `nx=ny=nz=20` which gives 32000 atoms. When `nx=ny=nz=200` CoMD simulates 32 million atoms which is half the size of the HIV virus molecular structure.

CoMD may execute multiple iterations per second, a typical number being several dozens. To analyze the system dynamics, the output results need to be treated in a timely manner. As developing the analysis treatments as a parallel application requires high technical expertise, the HPC community turns to using "big data" processing tools. The expectations are to succeed in providing efficient data analysis treatments while taking advantage of simpler programming models.

In this context, we consider two online data analysis treatments of the CoMD simulation results. The first computes a position histogram of atoms. The simulation area being represented as a 3D grid, the histogram reflects the number of atoms in each cell at each iteration. The second analysis is more computation intensive as it computes the travel distance of atoms between two iterations.

Both analysis applications are developed as Storm topologies with one spout and one analysis bolt. The spout injects the simulation results into the application, while the bolt computes the analysis. As there may be important queueing issues while transferring data from the HPC simulation to the Storm analysis treatment, we simulate the arrival of data at the spout level.

Our reactive elasticity strategy has succeeded in the automatic dimensioning of the Storm analysis application as a function of the CoMD simulation size (Fig. 11). We have considered the default size problem and have varied the injected load. We have started at 10 iterations per second (*it/s*) and increased the rate until 1000 it/s. In size, 1000 it/s for the default problem (32 thousand atoms) is equivalent to 1 it/s for the real-size problem (32 million atoms).

Before being able to apply the elastic reconfiguration to the bolt using the `health` metric, we faced the problem of the spout not being able to inject enough tuples into the system. To solve this issue, we adapted our bolt-oriented strategy to consider the spout activity. Instead of considering the ratio between the received and executed tuples (bolt), we considered the ratio between the emitted tuples and the target workload (spout). It appears that one instance of the spout has difficulties in injecting more than 300 it/s as it saturates in memory. To be able to inject 500 it/s we need two spout instances which translates in provisioning two VMs. To reach 1000 it/s we need four spout instances and thus four VMs. We have investigated multiples of two as this naturally fits the CoMD problem decomposition.

As for the processing bolts, they are CPU-intensive and increasing the number of instances is done by increasing the number of executors, one 4-vCPU VM being able to support four executors. To analyze 10 it/s for the small problem (Fig. 11), we need one instance for the spout (1 VM, 1 executor), one instance of the histogram bolt (1 VM, 1 executor) and one instance (1 VM, 1 executor) of the distance bolt. To treat 1000 it/s (1 it/s for the 32 million atom problem), the numbers respectively jump to 4 spouts (4 VMs, 4 executors), 12 histogram bolts (3 VMs, 12 executors) and 24 (6 VMs, 4 executors) distance bolts.

6 Related Work

Elastic resource provisioning has extensively been considered in the domain of cloud computing [33], [34], [35]. Numerous projects have proposed different elasticity solutions in terms of resource scaling units and strategies for autonomic adaptation. It is now largely accepted that elasticity may be applied vertically, by changing the capacity of resource instances, or horizontally, by adapting their number.

If there are plenty of cloud-oriented elasticity solutions, elastic strategies for stream processing systems have started to be investigated quite recently. In [10], for example, the authors show the usefulness of elastic provisioning of virtual machines to meet the varying real-time processing needs of an IoT environment. In FUGU [16], scaling considers CPU-overloaded or CPU-underloaded hosts and uses operator migration and not replication.

Among the proposals for stream processing systems using operator parallelization, neither considers the provisioning of different types of resources. In [12], for example, the provisioning of new resources, required in the case of a bottleneck, is delegated to the user and could greatly benefit from our multi-level provisioning proposal. The authors solve the optimization problem of guaranteeing latency constraints with minimum resources i.e. with minimal parallelization. If in our approach we start with a minimal configuration and consider increasing parallelism degrees, the authors propose a gradient descent algorithm that first checks whether the optimization constraints can be fulfilled with a maximum scale-out before considering smaller configurations.

Several projects [19], [20], [36] investigate explicit state management when parallelizing stateful operators and are thus complementary to our work. StreamCloud [20], for example, defines an automatic parallelization of the join, cartesian product and aggregator operators. When the CPU usage is detected to be out of bounds, elastic reconfiguration protocols exchange control tuples to redefine the computing windows among operator instances, reroute and load-balance tuples. Resource provisioning is delegated to a resource manager and does not take into account different types of resources. A similar approach is developed in [36] where the authors propose a solution to automatically detect data-parallel regions and compute the level of parallelism needed for an operator. The provisioning of new resources is out of their scope. Finally, ChronoStream [19] defines mechanisms for tracking computation progress, as well for state checkpointing and migration. Contrary to the previously cited proposals, it explicitly addresses both vertical and horizontal scaling. Horizontal scaling consists in provisioning of new hosts by external resource managers, as well as in varying the number of execution containers (processes) per operator. As for vertical scaling, it manages the number of threads allocated to an execution container. ChronoStream does not however address the questions of when and how many resources to provision.

The increasing popularity of the Apache Storm stream processing framework has motivated multiple research efforts. We can distinguish between works that focus on improving the Storm scheduling algorithm and approaches

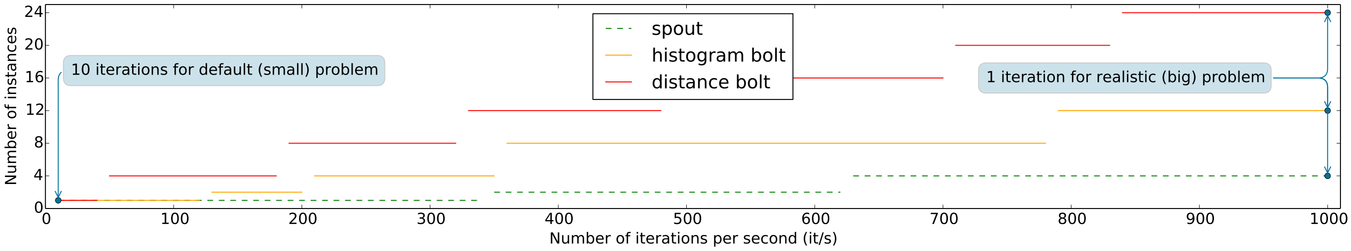


Fig. 11: Needed instances for online data analysis of CoMD simulation results: 1 spout instance requires 1 VM. 1 bolt instance requires 1 executor, a 4-vCPU VM supporting up to 4 executors.. The left blue point corresponds to 10 iterations of CoMD for 32000 atoms (small problem). The right blue points correspond to 1 iteration for 32 million atoms (real-size problem).

that explicitly consider elastic scaling. The projects of the first group tackle the problem of placing a *fixed* number of operator instances on a *fixed* set of resources. R-Storm [37], for example, proposes a resource-aware scheduling algorithm that takes into account CPU, memory and bandwidth usage. Using user-provided estimations of tasks' resource demands and analyzing the application topology, the algorithm aims at minimizing the network distance between communicating entities and at choosing the nodes with the best possible match for the tasks' resource demand. In [38], the authors generalize the formulation of the placement algorithm and show how it can be used for comparison of scheduling strategies. Finally, [39] pushes the scheduling effort further as it proposes an online scheduling algorithm that periodically adapts the placement according to the measured network traffic and resource usage. It is related to our multi-level resource provisioning as it differentiates the scheduling of Storm's executors and workers.

Similar to our work, the projects considering Apache Storm elasticity [13], [14], [40] target operator congestion and scale by adapting the parallelism degree of operator instances. In AUTOSCALE [13], for example, the system's scaling needs are deduced using both local and global estimations of the input workload and processing capacity. AUTOSCALE defines an activity level metric that aims at capturing the balance between the parallelism degree and the workload of an operator. Low activity levels call for scale-in operations, while high activity levels trigger scale-outs. In Stela [14] the scaling decisions are not automatic but requested by the user who defines the number of machines to add or to remove. Upon such a request, Stela chooses operators to scale using a global topology metric that reflects the operators' impact on the final throughput. The elasticity decisions adapt the number of provisioned executors for an operator so as to maintain the same number of operator instances (load-balancing) over the machines. In [40], the authors support our approach as they observe that the scaling of operators does not necessarily need to translate into the same resource scaling decisions. Focusing on CPU-bound applications, the authors use advanced application profiling and learning algorithms to establish thresholds triggering scaling. From the resource point of view, however, the three projects operate at the executor level and discard the aspects of hierarchical provisioning. Moreover, the configurations specifying the number of workers per machine and the number of executors per worker are decided *a priori* and in most cases in an ad-hoc manner.

7 Conclusion

The focus of our paper is on the impact of different execution containers on the performance of an elastic stream processing system. We have explicitly considered the hierarchy of execution containers (machines, processes and threads) and have shown that their provisioning comes at a different cost. More importantly, we have shown that provisioning the wrong type of containers may decrease performance.

We have proposed an elastic resource management for the Apache Storm system which scales an application while using the cheapest resource configuration. Our implementation is transparent to applications as it preserves the Storm API. As shown with two real-world applications, our system succeeds in maintaining the end-to-end latency of applications by detecting and healing local bottlenecks. The minimal execution configurations to support given levels of workload are discovered during a preliminary phase of application benchmarking. This approach is straightforward yet efficient in reflecting both the specific execution environment and application needs. In our future work we are interested in integrating explicit models of any preliminary knowledge about system.

An exciting perspective is to bring together existing advanced elasticity solutions [15], [36] and our work on multi-level containers. On one hand, existing solutions need to account for the underlying execution hierarchy. On the other hand, our proposal for a preliminary benchmarking may be too costly or even impossible for some applications. It would be interesting to integrate a control loop which dynamically integrates new facts about the current execution and adapts the elasticity management accordingly.

Future developments will also focus on the largely spreading container technologies such as Docker [41] and Singularity [42]. Elastic management could benefit from their lightweight management but would need to account for performance interference. To advance towards generic elasticity control, the way to go is to explicitly dimension execution containers. Indeed, in Storm there is already the possibility to define resource requirements for application operators. In Heron [8], [43], proposed as a successor of Storm, application operators are isolated as deployed using containers like those managed by Mesos [44]. The possibility to explicitly define the resource usage of execution containers brings, however, other challenges. Applications would need to be well-profiled to discover the resource needs of different operators. To prevent resource over-provisioning at the container level, application components would need to be deployed in containers with different capacities which in turn call for multi-dimensional-bin-packing-oriented scheduling [45].

Acknowledgements

The experimental work presented in this paper would not have been possible without the existence of the Grid′5000 platform and the help of the supporting teams. The authors would also like to thank the *enos* team who made the Openstack deployment process a child's play.

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] D. Duellmann. (2015) Big data and storage management at the large hadron collider.

[3] H. Yu, C. Wang, R. Grout, J. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *Computer Graphics and Applications, IEEE*, vol. 30, no. 3, pp. 45–57, May 2010.

[4] R. . Young, "Big data: changing the way businesses compete and operate," http://www.ey.com/Publication/vwLUAssets/EY_-_Big_data:_changing_the_way_businesses_operate/\$FILE/EY-Insights-on-GRC-Big-data.pdf, 2018.

[5] Oracle, "Oracle fast data: Real-time strategies for big data and business analytics," http://www.oracle.com/us/solutions/fastdata/fast-data-gets-real-time-wp-1927038.pdf, 2013.

[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[7] "Apache Storm," https://storm.apache.org.

[8] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *SIGMOD Conference*, 2015.

[9] "Amazon Kinesis," http://mesos.apache.org/.

[10] C. Hochreiner, M. VÃűgler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016.

[11] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 276–287.

[12] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, 2015, pp. 399–410.

[13] R. Kotto-Kombi, N. Lumineau, and P. Lamarre, "A Preventive Auto-Parallelization Approach for Elastic Stream Processing," in *37th (IEEE) International Conference on Distributed Computing Systems*, ser. 37th (IEEE) International Conference on Distributed Computing Systems, I. C. Society, Ed., Atlanta, United States, Jun. 2017, pp. 1532–1542.

[14] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, April 2016.

[15] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:14. [Online]. Available: http://doi.acm.org.ins2i.bib.cnrs.fr/10.1145/2038916.2038921

[16] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 13–22.

[17] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[18] "CoMD," https://gpuopen.com/compute-product/comd/.

[19] Y. Wu and K. L. Tan, "ChronoStream: Elastic Stateful Stream Computation in the Cloud," in *2015 IEEE 31st International Conference on Data Engineering*, April 2015, pp. 723–734.

[20] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.

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

[22] OpenStack. https://www.openstack.org/.

[23] "Grid′5000," http://www.grid5000.fr/.

[24] R. Cherrueau, D. Pertin, A. Simonet, A. Lebre, and M. Simonin, "Toward a Holistic Framework for Conducting Scientific Evaluations of OpenStack," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017.

[25] "Multi-Language Support in Apache Storm," https://storm.apache.org/about/multi-language.html.

[26] "Apache Thrift," http://thrift.apache.org/.

[27] "The Ganlgia Monitoring System," http://ganglia.info/, 2016.

[28] I. Safieddine, "Optimisation d'Infrastructures de Cloud Computing dans des Green Datacenters," Ph.D. dissertation, UGA, 2015.

[29] "eolas Group Business & Decision," http://www.eolas.fr/.

[30] P. Huber, *Robust Statistics*, ser. Applied Probability and Statistics Section Series. Wiley, 2004.

[31] "Lambda Architecture," http://lambda-architecture.net/.

[32] "CoMD," https://github.com/ECP-copa/CoMD.

[33] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 23–27.

[34] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Mos: Workload-aware elasticity for cloud object stores," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 177–188. [Online]. Available: http://doi.acm.org.ins2i.bib.cnrs.fr/10.1145/2907294.2907304

[35] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça, "MeT: Workload Aware Elasticity for NoSQL," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 183–196. [Online]. Available: http://doi.acm.org.ins2i.bib.cnrs.fr/10.1145/2465351.2465370

[36] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic Scaling for Data Stream Processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, Jun. 2014.

[37] 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. New York, NY, USA: ACM, 2015, pp. 149–161.

[38] 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 (DEBS)*. New York, NY, USA: ACM, 2016.

[39] 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.

[40] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 572–585, March 2018.

[41] "Docker," https://www.docker.com/.

[42] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017.

[43] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang, "Twitter Heron: Towards Extensible Streaming Engines," in *IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017.

[44] "Apache Mesos," http://mesos.apache.org/.

[45] O. Beaumont, J. Lorenzo, L. Eyraud-Dubois, and P. Renaud-Goud, "Efficient and robust allocation algorithms in clouds under memory constraints," in *International Conference on High Performance Computing (HiPC)*, Dec 2014, pp. 1–10.