Towards Vertically Scalable Spark Applications

Luciano Baresi and Giovanni Quattrocchi(✉)

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria,
Piazza Leonardo Da Vinci 32, Milan, Italy
{luciano.baresi,giovanni.quattrocchi}@polimi.it

Abstract. The dynamic provisioning of virtual machines (VMs) supported by many cloud computing infrastructures eases the scalability of software applications. Unfortunately, VMs are relatively slow to boot and public cloud providers do not allow users to vary their resources (vertical scalability) dynamically. To tackle both problems, a few years ago we presented a solution that combines the management of VMs with the use of containers specifically targeted to the efficient runtime management of the resources provisioned to Web applications. This paper borrows from this solution and addresses the problem of provisioning resources to big data, Spark applications at runtime. Spark does not allow for the runtime scalability of the resources associated with its executors, but resources must be provisioned statically. To tackle this problem, the paper describes a container-based version of Spark that supports the dynamic resizing of the memory and CPU cores associated with the different executors. The evaluation demonstrates the feasibility of the approach and identifies the trade-offs involved.

Keywords: Containers · Big data · Spark · Resource allocation

1 Introduction

The virtualization and softwarization of computing resources fostered by cloud computing has made the on-demand allocation/deallocation of computing means extremely easy. One can smoothly provision virtual machines (VMs) dynamically to cope with different workloads and meet stated qualities of service and/or cost constraints [7]. Many approaches [10,15] use different techniques to foresee and modify the number of allocated VMs properly and smartly, but unfortunately, Mao et al. [11] demonstrate that a VM on a public cloud infrastructure takes on average six minutes to boot. This is a too long delay when one thinks of the dynamic provisioning of resources to modern applications: for example, if the

This work has been partially supported by the GAUSS national research project (MIUR, PRIN 2015, Contract 2015KWREMX) and by project EEB (Italian Technology Cluster For Smart Communities, CTN01_00034_594053).

G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 106–118, 2019.
https://doi.org/10.1007/978-3-030-10549-5_9

workload increases, users do not want to wait for six minutes before being able to interact with the system properly. Instead of booting new VMs, one could alternatively think of adding resources to running VMs, but this is not possible since VMs usually come in fixed configurations and the resources associated with them (vertical scalability) cannot be changed. To overcome these two problems, that is, the latency of newly provisioned resources and the resizing of running ones, we proposed a solution [4] that pairs VMs and containers [16] for the fast and fine-grained allocation of resources to web applications. The idea is to deploy containerized web applications in a cluster of VMs, where each container is equipped with a lightweight control-theoretical planner to quickly (i.e., in a few seconds) provision and scale (vertically) the resources associated with it.

Starting from these ideas, this paper addresses a different, but similar problem: the dynamic provisioning of resources for big data applications. These applications are batch applications executed on top of special-purpose frameworks, which slice input data and carry out the computation on each slice by means of parallel processes executed on a distributed cluster of (virtual) machines. Specifically, we address Spark [19] applications, since Spark is the most widely used framework for big data applications: it is more flexible than Hadoop [2] and can support more complex computations. It uses a master-slave architecture, and multiple distributed *executors*—Java processes dedicated to data processing—are deployed onto the cluster. The response time of these applications is defined as the time they take to process the entire set of inputs; resources are usually estimated to meet *deadlines*, that is, thresholds on response times [18].

Spark does not allow one to specify deadlines and allocates resources (i.e., CPU cores and memory) to executors statically at the beginning of the execution; by default it always uses all available resources. This means that the resources that are allocated to applications must be planned carefully since runtime deviations are not allowed. The only dynamism managed by Spark refers to switching off preallocated executors if they remain idle for a user-defined amount of time, and on again if some tasks have to wait for too long (and idle executors are available), respectively. In addition, the resources provisioned to executors (e.g. CPU cores) cannot be changed. The scalability is only horizontal and based on simple time-outs, and on the availability of preallocated executors.

In contrast, this paper discusses and evaluates the feasibility of adding vertical scalability to Spark executors. It presents xSpark[1], a container-based extended version of Spark that allows for the fine-grained allocation of resources (CPU cores and memory) to applications, and that also supports the vertical scalability of executors (containers).

The rest of the paper is organized as follows. Section 2 surveys what industry tools offer in terms of dynamic resource allocation and introduces some related work. Section 3 motivates the need for vertical scalability and the use of containers as enabling technology. Section 4 describes the architecture of xSpark and how it supports the dynamic allocation (vertical scalability) of both CPU

[1] This paper extends [5] with an in-depth description of the technical details of xSpark that enable the vertical scalability of resources.

cores and memory. Section 5 presents the assessments we carried out and Sect. 6 concludes the paper.

2 Related Work

Spark only provides limited functionality to adjust the resources allocated to applications. By default, and at each execution, Spark always uses all the resources in the cluster. This means that when applications are running, and a new application is submitted for execution, it must wait since all resources are already taken. Alternatively, at submission time, Spark offers three parameters for allocating a smaller amount of resources to a specific application, leaving resources available to other subsequent application. Parameter `total-executor-cores` sets an upper bound to the total amount of CPU cores that an application can use, while parameter `num-executors` sets the number of executors. Therefore, the ratio between these two parameters gives the average number of cores allocated to each executor. Finally, parameter `executor-memory` sets the memory allocated to each executor.

The memory and cores allocated to executors cannot be changed at runtime since the vertical scalability of executors is not supported by Spark. However, Spark offers a *dynamic resource allocation* mode—governed by parameter `spark.dynamicAllocation.enabled`—to scale the number of executors at runtime. At submission time, parameter `spark.dynamicAllocation.initialExecutors` is used to set the initial amount of executors (instead of parameter `num-executors`) and parameters `spark.dynamicAllocation.min|maxExecutors` set the allowed range. To scale the number of executors Spark uses a simple heuristic based on utilization: if an executor remains idle for a predefined amount of time, it is decommissioned. If idle executors exist and a task remains pending for too long, a new executor is commissioned using a backlog algorithm: the first time Spark allocates an executor, if another request is triggered shortly (yet another parameter) the number of allocated executors is doubled, and so on. These time-outs are set statically and cannot vary at runtime.

As for additional resource management solutions, Spark can be paired with external resource managers—such as Mesos [8] and YARN [17]. Mesos sends resource offers (*push-based* scheduler) to its clients and manages both CPU cores and memory, while YARN waits for resource requests (*pull-based* scheduling) and only considers memory (each executor is bound to a single core). They both support containers to launch executors, but they do not offer any form of vertical scalability. Mesos also provides an optional fine-grained mode, where each task is containerized, but the runtime overhead is heavy, and this is why the use of this feature is deprecated in Spark 2.0.

xSpark offers two major improvements with respect to both Spark alone and Spark equipped with Mesos or YARN. First, it supports dynamic resource provisioning with respect to deadlines. This is not possible with existing industrial tools that only scale resources according to the utilization of the system. Second,

it supports the vertical scalability of executors with respect to both CPU cores, by means of containers, and memory, through the use of off-heap memory. This allows one to be precise and fast when scaling resources and also to minimize the overhead needed for creating/destroying executors. As for Mesos and YARN, xSpark is complementary to them: it is built on top Spark alone and we plan to extend our control capabilities to Mesos and YARN in the future.

Even if they do not target Spark specifically, it is also worth mentioning few works that exploit containers to provide the vertical scalability of resources [3,14]. Lakew et al. [9] rely on Linux containers to build fast and fine-grained controllers for the management of multiple resources. They exploit vertical scalability to meet performance indicators while optimizing resources for both interactive and non-interactive applications. Barna et al. [6] propose a methodology to build autonomic systems for containerized multi-tier applications. They exploit layered queuing networks to create self-tuning controllers for applications composed of heterogeneous components such as web services, databases, and big data elements. These solutions could be used to manage the resources allocated to a complete Spark instantiation, but they cannot manage the resources allocated to the different applications since they have no visibility of them. xSpark can do that since besides working on dynamic resource management, we have also changed the architecture and processing model behind Spark to work at a lower granularity level.

3 Vertical Scalability with Containers

The advent of cloud computing infrastructures has significantly simplified the runtime management of computing resources, and solutions from both industry [1] and academia [13,15] have proliferated. These solutions use virtual machines (VMs) to change the amount of CPU cores and memory allocated to applications and fulfill set quality requirements. Public cloud providers however only provision virtual machines with a fixed amount of memory and CPU cores. VMs can simply be created or deleted, and thus only the *horizontal scalability* of resources is supported. *Vertical scalability*, that is, the capability of modifying the amount of resources associated with a VM while it is in operation, is limited by the fact that users have no access to the hypervisor.

Another limitation of VM-based resource management is that it is too slow. According to Mao et al. [11], cloud providers require some 6 min to launch a new VM. Since new resources cannot be allocated faster than that, this *delay* imposes stringent limitations on how frequently allocated resources can change, and thus on how quickly these systems can meet user expectations.

These problems can be solved by adopting containers [12] as means to manage resources since they can be launched faster than VMs and can support both horizontal and vertical scalability. Containers can be booted in few seconds (depending on the application type) and scaled vertically in hundreds of milliseconds [16]. Resource managers can then be as fast as their actuators and adopt control periods that are less than a second: this is the case of the control-theoretical solution presented in Sect. 4.

In addition, VMs are usually dedicated to only one process at a time since the simultaneous execution of independent application components cannot be easily managed, and unexpected resource contention may arise. With containers instead, a single machine can be used more efficiently to deploy multiple parts of the same application—or of different applications—since each container is provisioned individually and is isolated from the others.

Containers alone are not sufficient to making Spark executors scale vertically. xSpark exploits Docker containers to achieve vertical scalability and embeds then into Spark to scale the number of CPU cores allocated to executors at runtime. However, memory allocation is limited by the Java Virtual Machine (JVM), which sets a static upper bound to allocated memory at startup time, and this value cannot be changed without restarting the virtual machine itself. For this reason we extended Spark to allow for the dynamic resizing of *off-heap* memory, as discussed in Sect. 4.3. xSpark exploits vertical scalability to control the execution time of Spark applications in order to allocate resources efficiently and fulfill user deadlines. A similar result could be achieved through horizontal scalability, but with less efficiency since vertical scalability is faster and works at a finer level.

4 xSpark

xSpark[2] extends Spark and uses containers (i.e., Docker) to support a more flexible and advanced management of resources. xSpark enriches command `submit` with an additional parameter `deadline` to specify the required execution time. Since the goal of xSpark is to minimize the use of resources without violating deadlines, xSpark interprets this input as a constraint on execution: finishing before the deadline would mean that fewer resources could have been used, while violating it means that too few resources are provisioned (or are available).

This section describes xSpark atop virtual machines, but our tool can also be deployed on bare-metal to favor performance over flexibility.

4.1 Hierarchical Architecture

Figure 1 shows the master/slave architecture of xSpark: white boxes represent the existing components we modified, gray boxes are container-related components, and dark-gray boxes correspond to new, control-related components.

The *Master Node* hosts a *Stage Scheduler*, a *Task Scheduler*, and a heuristic-based *Application Level Controller* for each running application (Fig. 1 assumes the existence of two applications). Spark logically splits applications into *stages*[3], a key entity for xSpark. In fact, xSpark has modified component *Stage Scheduler* to intercept the beginning and end of each stage and uses the heuristic embedded in *Application Level Controller* to compute an execution *deadline* for each stage.

[2] The source code of xSpark is available at https://github.com/deib-polimi/xSpark.

[3] A Spark stage is a set of pipelined operations that do not require shuffling data among nodes.

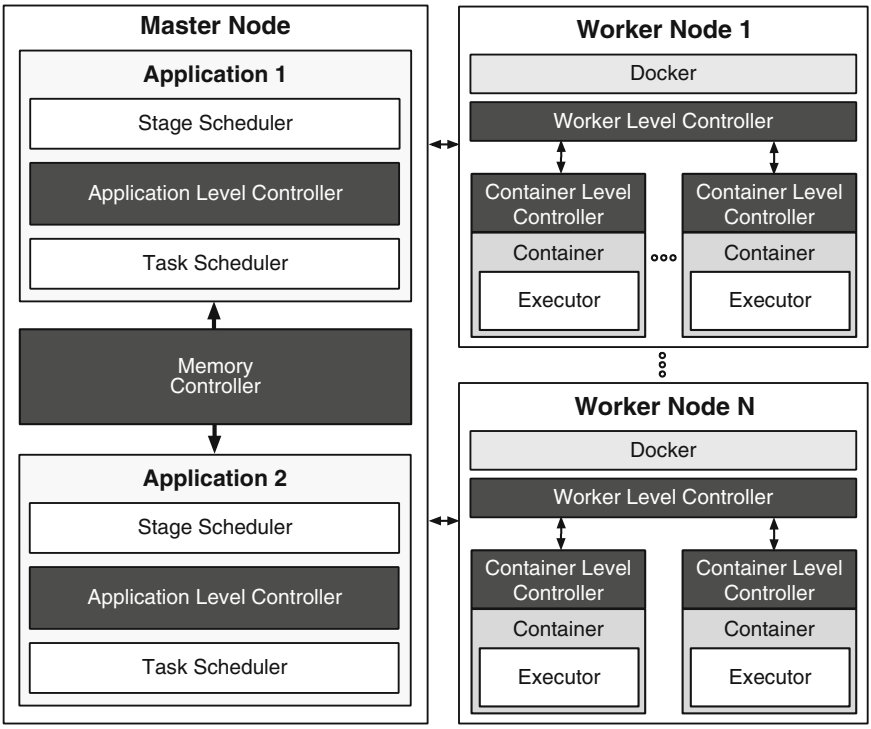

Fig. 1. High-level architecture of xSpark.

The heuristic considers the remaining amount of time, with respect to the global deadline set at application level, and some performance data collected through a profiling phase. Since xSpark needs to know the internals of each stage, this preliminary activity is used to create the actual execution flow (direct acyclic graph) of each stage and some performance metrics (e.g., the duration of each stage, number of input/output records).

After estimating execution deadlines, the actual execution of an application's stages start in the different worker nodes. Since stages can be composed of diverse operations, we advocate that resource allocation should be controlled at stage level (and not at application level). Therefore, xSpark executors are dedicated to single stages, while Spark executors can execute the tasks of any stage. This way, the resources (dynamically) provisioned to a given executor can only impact the performance of the stage associated with it, and xSpark can even obtain a finer-grained control of the execution of the different stages, and thus of the whole application. This allows xSpark to control stages individually and to equally distribute computation and data over the whole set of nodes.

As described in Sect. 4.2, executors are wrapped in containers and individually controlled by a control-theoretical planner (*Container Level Controller*). The planner uses CPU quotas to provide computing resources to make the

execution times of stages last as close to estimated deadlines as possible. Note that the planners bound to the executors dedicated to the same stage do not need to be synchronized since they are configured to fulfill the same deadline and the workload (i.e., the tasks to be processed) is equally split among them. The planners exploit a feedback loop that monitors the progress of the executors (i.e., number of completed tasks over the total) and allocates processing power (i.e., CPU quotas) accordingly.

Finally, resource contention within a *Worker Node* could occur because different executors bound to specific applications/stages are deployed onto the same machine. For this reason, xSpark uses a *Worker Level Controller* that gathers all the core allocation requests from the control-theoretical planners and, if their sum is greater than available cores, scales them down according to different configurable strategies, such as Earliest Deadline First or proportionally.

4.2 CPU Cores

Vertical scalability is the key feature provided by xSpark: it enables continuous control over executing applications by managing allocated resources without restarting/deallocating containers, thus reducing the associated overhead. While memory is either sufficient, and any increase would produce no benefit, or insufficient, provisioned computing resources, given the high degree of parallelism embodied in Spark applications, can significantly impact execution time: the more CPU cores one allocates, the faster the application should execute.

Spark deploys executors onto *Worker Node*s (virtual/physical machines) and uses Java Virtual Machines to execute them. The allocation of CPU cores is static and managed by a simple, internal, pool of threads. xSpark instead deploys each executor in a container by using *Docker* to allow xSpark to dynamically change the computing resources provisioned to an executor without interfering with the other executors running on the same node.

Docker provides three ways to allocate CPU cores dynamically: reservations, shares, and quotas. All these methods are extremely fast (few hundreds of milliseconds [16]), but they heavily differ in terms of granularity and reliability. Reservations allocate specific cores to containers. For example, given a machine with 8 cores, *container1* can be pinned to the first 5 cores, while *container2* to the remaining 3 cores and cannot use any of the cores allocated to *container1* if not used. CPU reservation is deterministic since each container can only use the cores allotted to it while granularity is limited to full cores.

With shares, each container uses at least a number of cores that is proportional to its *shares* but if there is no contention, and additional cores are available, a single container can even exploit all available cores. For example, if *container1* has 70 shares and *container2* 30 shares, in case of resource contention *container2* uses some 70% of the cores of the machine while *container1* just 30%. However, if *container2* only needed 20% of available cores, *container1* could use the remaining 80%. Thus, shares are not always deterministic, since the actual number of used cores depends on both set shares and the number of

cores used by each container. The solution is extremely fine-grained since it can allocate fractions of cores.

Quotas provide the most powerful way of provisioning cores to containers and guarantees both determinism and appropriate granularity. Each container is associated with a *period* and a *quota*, where the latter represents the percentage of CPU time allocated to the container within the period. Setting a quota larger than the period means that the container should use more than one core at a time, but the sum of all quotas must always be less than set period times the number of available cores. For example, given a single-core CPU and a period of 100 ms, if *container1* is given a quota of 30 ms and *container2* a quota of $70ms$, then 70% of the CPU time is reserved for *container2* and 30% for *container1*. This mechanism is thus deterministic and very fine-grained.

xSpark associates each executor (container) with a control-theoretical planner that computes the amount of CPU cores needed at each control period. Since the faster, finer-grained, and more deterministic actuation capabilities are, the more precise these planners can be, xSpark uses quotas as allocation mechanism and associates all containers with the same period.

4.3 Memory

One of the main problems when executing a Spark job is how to determine the amount of memory to allocate to each executor. Spark allows one to specify this value statically by using parameter `spark.executor.memory`, which changes the size of the heap memory of all executors. When the heap memory of an executor gets saturated, the process crashes and the JVM is restarted.

When executing multiple applications, if their number is known, one can simply equally partition available memory to the applications, that is, $h = \frac{M}{|A|}$, where A is the set of running applications, h is the amount of heap memory allocated to an application $a \in A$, and M is the total memory available. Unfortunately, the number of applications to execute and when to execute them are often not known a priori, and thus the amount of memory associated with each executor inevitably impacts the maximum number of applications that can be run in parallel. This were not a problem if the heap memory could be scalable vertically, but unfortunately JVM's do not allow one to resize it at runtime: a given configuration can only be changed by restarting the JVM. Note that Spark postpones the launch of an application if requested memory cannot be provided.

To solve this problem, xSpark uses *off-heap memory* to add flexibility and be able to change memory boundaries dynamically. Although on-heap memory offers better performance, Spark can use off-heap memory to both support execution and store data. Objects stored in off-heap memory are managed directly by the operating system, are not part of the process heap, and are not garbage collected. As said, accessing off-heap data is slightly slower than on-heap data, but it is faster than reading and writing from/to disk (see Sect. 5).

Since Spark *does not* provide any means to resize the memory used by off-heap objects at runtime, xSpark offers a *Memory Controller*, which is deployed on the *Master Node*. Each executor is associated with a fixed quantity of on-heap

memory plus a quota of off-heap memory that can then be adjusted at runtime. This quota is decremented when a new application is submitted for execution and is incremented when an application terminates.

5 Evaluation

This section describes the experiments we conducted to evaluate the solutions we conceived for the dynamic allocation of both CPU cores and off-heap memory. The assessment is based on the following three questions: (Q1) is the vertical scalability of cores appropriate for controlling the response time of Spark applications? (Q2) what can we achieve by vertically scaling resources? (Q3) how does the use of off-heap memory impact the performance of xSpark when compared against the use of on-heap memory?

5.1 Vertical Scalability of CPUs

To answer Q1 we used two applications taken from the SparkPerf[4] benchmark suite: *sort-by-key* and *aggr-by-key*. These applications perform simple aggregation and sorting operations over a randomly generated dataset. We executed them on a single AWS EC2 *m4.4xlarge* VMs with 8 CPUs[5] and 64 GB of memory. We executed each application with 8 different configurations, and repeated each experiment three times, for a total of 24 executions. We started by allocating 1 core to each application. Then, we randomly changed the number of allocated cores every second by using a uniform distribution in the range between 1 and 3 to get an expected average value of 2 cores. As for the other experiments, we kept changing the number of allocated cores every second, and we kept increasing the expected value from 2 to 8 cores.

Figure 2 shows the results of our experiments; the blue dotted line renders the duration of the execution and refers to the left-hand y-axis, while the red crossed line corresponds to the speedup (over one core) and refers to the right-hand y-axis. The charts witness that vertically scaling the number of CPU cores assigned to an executor strongly impacts the response time of Spark applications with a close to linear speed-up.

To answer Q2 we studied how xSpark exploits vertical scalability to control the execution time of applications. To do that we tried to control the two aforementioned applications and a more complex one called *PageRank*, a graph-based algorithm that was taken from another benchmark suite called SparkBench[6]. We executed the three applications with different deadlines and datasets and we obtained an error as low as 1%, where the error was computed as $(deadline - actualDuration)/deadline$. An error equals to 0 means that xSpark was able to allocate the minimum amount of CPU cores and fulfill the deadline.

[4] Available at https://github.com/databricks/spark-perf.
[5] 8 cores without hyperthreading or 16 virtual cores if enabled.
[6] Available at https://github.com/CODAIT/spark-bench.

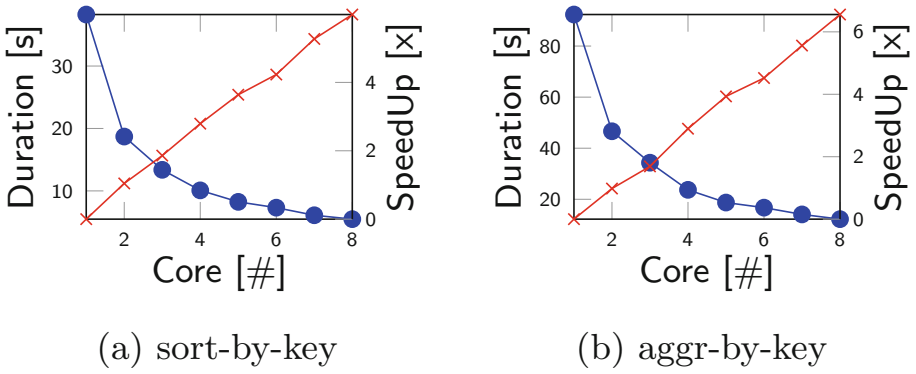

Fig. 2. CPU allocation over application duration (dots represent execution times while crosses refer to speedups).

To better visualize how xSpark works Fig. 3 shows the details of a controlled executor while processing PageRank. Note that during its life-cycle an executor can execute different stages in a row (in this case 9 stages). In this chart, the black and gray lines, which refer to the left-hand y-axis, show the actual stage completion percentage (i.e., number of tasks completed over the total) and the imposed one, that is the set-point of the control theoretical planners. The blue line, which refers to the right-hand y-axis, shows the core allocated to the executor. The E-labeled green vertical lines represent stage ends (actual stage durations), while the red dashed vertical lines represent stage deadlines (the last one correspond to the application deadline). The fast and fine-grained vertical scalability provided by containers allows xSpark to make all the executors (closely) follow the prescribed progress rates for each stage and thus terminate the execution very close to the foreseen deadline (Table 1).

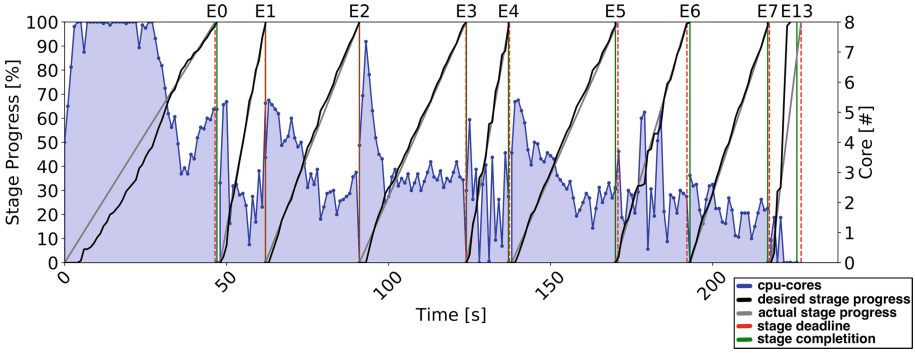

Fig. 3. Controlling PageRank.

Table 1. Off-heap impact.

App	On-heap	Off-heap	CPUTime	Delay
aggr-by-key	100%	0%	5166	-
	50%	50%	5502	6%
	10%	90%	6289	21%
PageRank	100%	0%	2013	-
	50%	50%	2051	2%
	10%	90%	2150	7%

5.2 Vertical Scalability of Off-Heap Memory

To answer Q3 we compared the performance of *aggr-by-key* and *PageRank* when only using on-heap memory or on-heap and off-heap memory. To carry out these experiments we used *Standard_D14_v2* VMs provided by Microsoft Azure, each of them had 16 CPUs, 112 GB of memory. The applications were executed with three distributions of on-heap and off-heap memory with fixed core allocation: *all on-heap* (100/0), *balanced on-heap and off-heap* (50/50) and *almost all off-heap* (10/90). To evaluate the differences among the different memory configurations, we rely on metric $CpuTime$, that is, the execution time times the number of used cores: needless to say, the higher this value is, the worse it is. Moreover, *aggr-by-key* was configured to use off-heap memory only for processing, while PageRank was instructed to use off-heap memory for both processing and storing data.

In the case of *aggr-by-key*, when decreasing the on-heap memory allocated to executors, up to 90%, $CpuTime$ increased by 21% (from 5166 to 6289). This significant difference is caused by disk swapping since cached datasets were persisted onto disk. In contrast, when running *PageRank*, which used off-heap memory also for storing data, the impact of disk swapping was dramatically reduced ($CpuTime$ only increases by 7%). This is in many cases a negligible performance reduction given that the use of off-heap memory allows xSpark to vertically scale the memory allocated to executors and foster the parallelism among applications.

6 Conclusions

This paper introduces xSpark, our extension to Spark that supports the vertical scalability of the resources allocated to executors. xSpark exploits containers to provide the dynamic, fast, and fine-grained allocation of cores to containers, and off-heap memory to allow for resizing the memory associated with executors. Our preliminary assessment shows that xSpark can control the execution time of Spark applications precisely and that the use of off-heap memory has limited impact on the execution time of applications.

References

1. Amazon EC2 Autoscaling. https://aws.amazon.com/autoscaling/
2. Apache Hadoop (2017). http://hadoop.apache.org
3. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Autonomic vertical elasticity of Docker containers with ELASTICDOCKER. In: IEEE 10th International Conference on Cloud Computing (CLOUD), pp. 472–479 (2017)
4. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 24th ACM International Symposium on Foundations of Software Engineering, pp. 217–228. ACM (2016)
5. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: Fine-grained Dynamic Resource Allocation for Big-Data Applications. Technical report (2018). http://hdl.handle.net/11311/1057275
6. Barna, C., Khazaei, H., Fokaefs, M., Litoiu, M.: Delivering elastic containerized cloud applications to enable DevOps. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017, pp. 65–75. IEEE Press (2017)
7. Dustdar, S., Guo, Y., Satzger, B., Truong, H.L.: Principles of elastic processes. IEEE Internet Comput. **15**, 66–71 (2011)
8. Hindman, B., et al.: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI 2011, pp. 295–308. USENIX (2011)
9. Lakew, E.B., Papadopoulos, A.V., Maggio, M., Klein, C., Elmroth, E.: KPI-agnostic control for fine-grained vertical elasticity. In: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 589–598. IEEE (2017)
10. Liu, J., Shen, H., Narman, H.S.: CCRP: customized cooperative resource provisioning for high resource utilization in clouds. In: Proceedings of the 3rd IEEE International Conference on Big Data (Big Data), pp. 243–252 (2016)
11. Mao, M., Humphrey, M.: A Performance study on the VM startup time in the cloud. In: Proceedings of the IEEE 5th International Conference on Cloud Computing, pp. 423–430. IEEE (2012)
12. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. (2014)
13. Nikravesh, A.Y., Ajila, S.A., Lung, C.H.: Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In: Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 35–45. IEEE Press (2015)
14. Rao, J., Bu, X., Xu, C.Z., Wang, K.: A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In: IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 45–54. IEEE (2011)
15. Seracini, F., Menarini, M., Krueger, I., Baresi, L., Guinea, S., Quattrocchi, G.: A comprehensive resource management solution for web-based systems. In: Proceedings of the 11th International Conference on Autonomic Computing (2014)
16. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, vol. 41, pp. 275–287. ACM (2007)

17. Vavilapalli, V.K., et al.: Apache Hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing. ACM (2013)
18. Verma, A., Cherkasova, L., Kumar, V.S., Campbell, R.H.: Deadline-based workload management for MapReduce environments: pieces of the performance puzzle. In: NOMS, pp. 900–905. IEEE (2012)
19. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd Conference on Hot Topics in Cloud Computing, HotCloud 2010. USENIX (2010)