A SLA-based Spark Cluster Scaling Method in Cloud Environment

Yoori Oh Jieun Choi Eunjung Song Moonji Kim Yoonhee Kim
Dept. of Computer Science
Sookmyung Women's University
Seoul, Korea
{yoori0203, jechoi1205}@sookmyung.ac.kr, {songfg78, km0420jj}@gmail.com, yulan@sookmyung.ac.kr

***Abstract*— As the development of Internet and mobile device increases, there is a correspondingly increasing amount of data produced by users of such technology worldwide. It is thus essential to analyze such massive amounts of data reflective of the big data era. Recently, Apache Spark has become popular for analyzing big data, since it can process streaming data and support real-time in-memory computing. Also, it is known to execute applications faster than traditionally used Hadoop. Also cloud technology provides flexible resource utilization environment on-demand. When analyzing big data using Spark in existing environments, it is difficult to provision resources according to the system's changing environment and the influence of other users' executions. Using cloud technology however, it is possible to provision resources more effectively for the execution of jobs through dynamic resource provision methods. In this paper, we propose an auto-scaling framework with corresponding algorithms to manage resources dynamically in virtual environments, in order to meet user-specified SLA (Service Level Agreement) given a set of limited resources. Our experimental results on Spark in OpenStack demonstrate the effectiveness of scaling resources to satisfy user SLAs.**

Keywords—SLA, auto-scaling, Spark Cluster, resource management,

I. INTRODUCTION

With the development of Internet and mobile devices, there is a correspondingly increasing amount of data produced by users of such technology worldwide. Accordingly, Apache Hadoop [1], initially appeared as distributed processing framework which supports big data analysis application which cannot be processed with a single machine using MapReduce algorithm. However, Hadoop [1] is known for having I/O bottleneck during the reading and writing of data to disk. Also it is unable to support real-time processing. Due to these drawbacks, Apache Spark [2], was recently developed to facilitate the processing of real-time streaming data and the application of in-memory computing. Big data analysis is difficult to process in batch-processing and usually consists of successive several stages running for long time periods. In this situation, it is essential that the system required to deal with various jobs in real-time should be able to manage resources dynamically.

Cloud technology provides suitable execution environment satisfying Spark's dynamic resource requirement, since it enables Spark to use more than available physical resources through virtualization. In this regard, it is crucial to apply proper resource auto-scaling to handle difficult situations such as various SLA (Service Level Agreement; e.g. deadline), dynamic resource requirement, and unexpected job submit time. Especially, Spark application users submit various requirements during the processing of big data, thus it is necessary for the user to provide scaling mechanism to meet their SLAs.

In this paper, we propose an auto-scaling method for utilizing resource of Spark clusters effectively in cloud computing environment. The proposed auto-scaling method has a goal to meet user-specified deadline. Also we perform experiments to verify the effectiveness of the proposed scaling algorithm.

This paper is organized as follows. Section 2 provides related work of Apache Spark and Auto-Scaling. Section 3 presents the service architecture of auto-scaling framework. Section 4 explains SLA based auto-scaling algorithm and we discuss experiment in Section 5. Finally, we conclude in Section 6.

II. RELATED WORK

A. Apache Spark

Spark [2] framework have emerged following existing Hadoop [1] framework, which was studied widely. Spark [2] supports distributed processing on several nodes which is similar to Hadoop and in-memory computing as major feature. Unlike Hadoop executing disk I/O for data processing, Spark provides in-memory computing using a new concept of data structure RDD (Resilient Distributed Dataset). It is useful for iterative execution or streaming data processing, thus it could derive execution result faster than Hadoop.

References [3]–[5] are studies for efficient execution of Spark. H. Chen et al. [3] suggest entropy concept based scheduling method to provide effective and reliable services when it provides online parallel analysis services. Entropy concept is used to measure disordering of system and it becomes the standard of decision for reliability. They also utilize dynamic core performance by scheduling their resources. J. Yin et al. [4] are a study of optimization for parallel access in

big data processing. They propose a method of matching storage server and parallel data request, and a plan to improve the performance in interactive data access method. R. Palamuttam et al. [5] suggest weather event detection and tracking system using Spark. They analyze scientific applications applying a new data structure, sRDD (scientific RDD), and demonstrates a feature of data reuse between stages. Sidhanta, Subhajit et al. [6] propose a model of job execution time on Spark [2]. The model is used to estimate the completion time of a given Spark job according to the size of the input datasets, the number of iterations, and the number of nodes in the cluster. It also estimates the cost optimal cluster composition for running a given Spark job on a cloud under a completion deadline.

B. *Auto-Scaling*

References [7]-[8] are studies of resource auto-scaling for satisfying user's requirements and efficient resource utilization. Kang, Hyejeong et al. [7] suggest auto-scaling method in hybrid cloud environment. They propose cost efficient scheduling method to execute scientific applications satisfying its deadline in private and public cloud environment. Ahn, Yoonsun et al. [8] are a study of auto-scaling method considering application pattern in hybrid cloud environment. It proved proper resource utilization considering job's deadline and the feature of applications. References [9]-[10] are studies about auto-scaling scheme to effectively improve cloud resource usage for Hadoop big-data framework. Gandhi, Anshul et al. [9] propose auto-scaling method in Hadoop clusters. It represents a general model for execution time, tunes some parameters to fit in each workload, and formulates the estimated finish time. It shows auto-scaling method based on that information. Jacob Leverich et al. [10] suggest scale-down of clusters deploying distributed big-data processing frameworks method that improve energy efficiency a Hadoop cluster. Xueying Wang et al. [11] present an auto-scaling approach for Hadoop system in private cloud in order to improve resource utilization. Also, they consider inference aware scaling method in the proposed multi-layer node model to reduce performance damage owing to conflict from other services. However, there was no auto-scaling method for Spark [2] clusters to satisfy SLA.

III. Service Architecture of Auto-Scaling Framework

In this section, we discuss architecture of our service framework. Figure 1 shows architecture of auto-scaling framework proposed in this paper. When a user submits a Spark application for execution with a given SLA (deadline). Spark application executes jobs using Openstack [12] as its resource. Openstack [12] is an open source software that provides large pools of compute, storage and networking resources used for the private cloud. It enables auto-scaling framework to dynamically scale resources on-demand.

'Auto-Scaling Service' plays a major role of auto-scaling framework, providing auto-scaling by monitoring in runtime. First, it monitors jobs in real-time whether it satisfies SLA which user submitted. When it is not possible to meet user's requirements, it provides the information of which application needs and how much it needs. This information is sent to run-time scaling service. The Scaling service makes decision on requested resource of each application in current situation. Decision information is sent to scheduling service and resources are scheduled for each application.

'Metadata Management Service' has aggregated information of jobs, resources and VM. Job metadata and resource metadata have information of profiling data created by executing jobs before. VM catalog consists of VM information of its cores, RAM and name. When the system needs to schedule resources, these information is used to check the satisfaction of SLA.

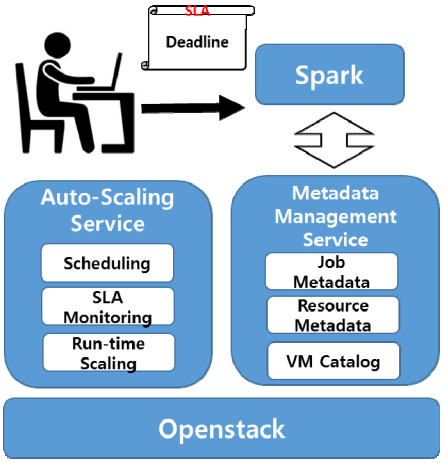

Fig 1. Architecture of Auto-Scaling Framework

IV. SLA based Auto-Scaling Algorithm

Algorithms are classified under two groups: Run-time Scaling (in Algorithm 1) and SLA monitoring (in Algorithm 2). For better understanding, main notations for the algorithms are explained at the below:

- AL= {a|a_i, i=1, 2, …, n}

  : An application list which is running
- D : Deadline of each application
- RVM : Number of running VM
- EFT : Estimated finish time
- TT : Number of total tasks
- ST : Start time of application
- α, β : percentage of execution time on its stage to total execution time
- SCALE_IN, SCALE_OUT = { [a, # of VM] | a ∈ AL}

  : # of VM is amount of VM that the application wants to scale in or out

  : Sets of an application candidate and its VM amount to scale-in and scale-out

Algorithm 1 describes run-time scaling method. It is invoked when both SCALE_IN and SCALE_OUT sets are not empty sets (line 2). When both sets have some elements, the

algorithm determines the number of applications to apply scaling to in each candidate application set. The number of VMs added or subtracted in each set for the first application are added to the number of total addition and subtraction VMs (line 3-4). And then it continues in a while loop until VMs for the last application in SCALE_IN or SCALE_OUT is adjusted (line 5-13). In the loop, the algorithm calculates the max addition or subtraction of VMs while it checks the VM numbers to add or subtract of each set in the while loop. After that, it sends information of VMs and applications to scale-in or scale-out to resource scheduling part (line 14). It waits for next interval to monitor the execution situation (line 15) and receives information periodically whether it needs scaling (line 16).

Algorithm 1. Run-time Scaling
Input: Running Application List *AL*,
Application Information = {the deadline *D*, number of running VM *RVM*, estimated finish time *EFT*, start time *ST*, the number of total tasks *TT* }
Output: Scaling decision S = {*SCALE_IN*, application index *i*, *SCALE_OUT*, application index *j*}

```
1: while (true)
2:     if (SCALE_IN !=∅ && SCALE_OUT !=∅)
3:         TotalAddVM = addVM_1 ;
4:         TotalSubVM = subVM_1 ;
5:         while (a_i or a_j is not the last application)
6:             if TotalAddVM > TotalSubVM then
7:                 i++;
8:                 TotalSubVM = TotalSubVM + subVMi;
9:             else
10:                j++;
11:                TotalAddfVM = TotalAddVM + addVMj;
12:            end if
13:        end while
           //send scaling decision to JES
14:        Scale(SCALE_IN, i, SCALE_OUT, j);
15:        waitForNextInterval();
16:        SCALE ← SLAMonitoring(AL, D, RVM, EFT, ST, TT);
17:    end if
18: end while
```

Algorithm 1. Run-time Scaling

Algorithm 2 shows monitoring algorithm based on SLA. It provides the information on which application needs how many VMs to scale-in or scale-out as a result. It calculates the number of estimated execution tasks using the number of total tasks, deadline for each application, execution ratio and execution time of that moment (line 3). α is the percentage of execution time on its stage to total execution time. It checks the number of estimated execution tasks against the number of processed tasks (line 4). When the condition is true, it is necessary to add VMs (line 6-9). And if the number of estimated execution tasks is smaller than the number of processed tasks, it requires a scale-in (line 12-15). Since EFT (estimated finish time) is the array of execution time according to the number of nodes in the cluster, index i means the number of nodes in the cluster. It calculates estimated finish time in case of applying to scale nodes (line 7, 13). It reflects the execution ratio of its stage and estimated finish time. And it determines the suitable cluster composition for finishing the job within user-specified deadline (line 8, 14). It returns the result in descending order of VM numbers in SCALE_IN and SCALE_OUT sets.

Algorithm 2. SLA Monitoring
Input: Running Application List *AL*
Application Information = {the deadline *D*, number of running VM *RVM*, estimated finish time *EFT*, start time *ST*, the number of total tasks *TT*}
Output: Decision of Scaling, *SCALE_IN*, *SCALE_OUT*

```
1: i ← 0;
2: for each application a in AL do
3:     EET = TT/(D* α)*ET ;
4:     if EET > NET then
5:         do
6:             i++;
7:             NEFT_i = EFT_i* α/ TT*(TT-NET) + EFT_i* β ;
8:         while(D -ET > NEFT_i)
9:         SCALE_OUT = SCALE_OUT ∪ {[ a, i- RVM ]};
10:    else if EET < NET then
11:        do
12:            i++;
13:            NEFT_i = EFT* α/ TT*(TT-NET) + EFT_i* β ;
14:        while(D -ET > NEFT_i)
15:        SCALE_IN = SCALE_IN ∪ {[ a, RVM-i ]};
16:    end if
17: end for
18: Sort SCALE_IN and SCALE_OUT in descending order of addVM and
                                                    subVM;
19: return SCALE_IN and SCALE_OUT;
```

Algorithm 2. SLA Monitoring

V. EXPERIMENTS

A. Preliminary Experiment

We compared execution times according to different number of workers when executing the same job. The job is a WordCount workload using a 5GB input data set. Worker VM has 4 cores and 8GB RAM, and we measured total execution time as the number of worker VM is increased for the map stage, reduce stage and the whole execution represented as total in Figure 4.

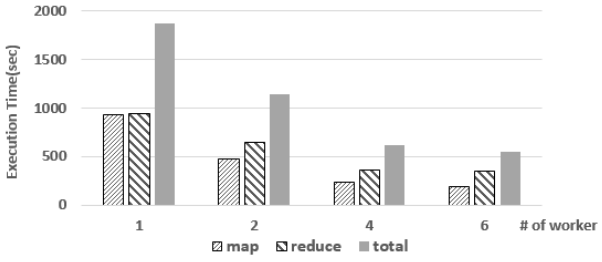

Fig 4. Execution Time for Different Number of Worker

Figure 4 shows the execution time for 4 different number of workers. As the number of workers is increased from 1 to 2, 2 to 4 and 4 to 6, the total execution time is reduced by 38%, 45% and 16% respectively. Also, the execution time of the job at the map stage is shorter than that of the reduce stage. As the number of worker is increased, the execution time of map stage

is reduced by 49%, 45%, 15% while that of the reduce stage is reduced by 31%, 44%, 21% respectively. Also in comparison, the ratio of execution time at the map stage relative to the reduce stage is 99%, 73%, 64%, and 55% respectively for the different number of workers. This shows that the more workers are added to the system, the more effectively the execution time of map stage is reduced. Our experiment reveals that scaling at map stage could get better result in execution time.

B. *Scaling Validation for Spark Clusters*

We verified the effectiveness of auto-scaling by executing a WordCount workload using 5GB input dataset according to different user-specified SLAs. The workload consists of 2 stage and each stage has 5278 tasks. The ratio of map stage to reduce stage is 1:2. We set the monitoring interval as 60s and preparation time as 30s. The goal is to scale the number of workers in the system in order to meet the user-specified deadline.

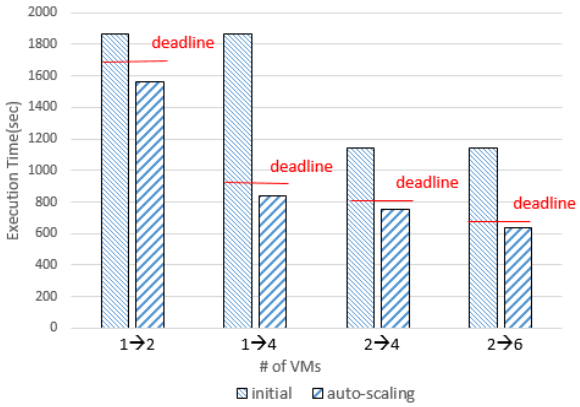

Fig 5. Satisfying User-specified SLA with Proposed Scaling Method

Figure 5 shows the performance of the proposed auto-scaling method comparing to initial scheduling. For initial scheduling method, we execute an application using 1 worker or 2 workers. On the other hand, the auto-scaling method, monitoring the execution of the application, performed scaling dynamically to finish within the given deadline. When the cluster is composed of 1 worker, we assign deadline as 1700s and 900s. In case of 1700s, the cluster adds 1 worker and the graph shows finishing within 1700s in Figure 5. When the deadline is 900s, the application finishes within 900s by adding 3 workers in the cluster. When the cluster has 2 workers, we assign deadline as 800s and 650s. The cluster adds 2 workers and 4 workers for each case. When the deadline is 800s, the system monitors how many tasks are processed at 90s for the first time because of the preparation time (30s) and monitoring interval (60s). The number of processed tasks is 230 and the number of estimated execution tasks calculated by the algorithm is 1187. Since the number of executed tasks are smaller than estimated execution tasks, it requires a scale-out. The algorithm determines the number of VMs to scale-out using profiling data of execution time on different number of worker VMs. If the cluster has 4 worker VMs, estimated finish time is 611s. On the other hand, the application can finish in 536s using 6 worker VMs. As the difference between the deadline and execution time is 710s, the plan of using 4 VMs is selected. And they finish their application within each deadline. Therefore, all cases are successfully finished within user-specified deadline using auto-scaling method.

VI. Conclusion and Future Work

With the exponential increase in the amount of data, more and more people have resulted to the use of Spark clusters to analyze big data. In the proposed scaling framework, Spark clusters facilitate the dynamic utilization of resources in virtual environments. In this paper we propose the auto-scaling framework with corresponding algorithms. The algorithms determine scaling decision and it calculates how many VMs to scale-in or scale-out. Our experimental results of deploying Spark clusters on OpenStack demonstrate the efficiency of auto-scaling for executing Spark clusters. In the future, we intend to research scale-out experiment to verify the effectiveness of auto-scaling in Spark clusters.

Acknowledgment

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (2015M 3C 4A7065646)

References

[1] Apache hadoop. [Online]. Available: http://hadoop.apache.org/.

[2] Apache Spark: Lightning-fast cluster computing, “Apache spark,” 2015. [Online]. Available: https://spark.apache.org/

[3] H. Chen and F. Z. Wang, "Spark on entropy: A reliable & efficient scheduler for low-latency parallel jobs in heterogeneous cloud," Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th, Clearwater Beach, FL, 2015, pp. 708-713.

[4] J. Yin and J. Wang, "Optimize Parallel Data Access in Big Data Processing," Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, Shenzhen, 2015, pp. 721-724.

[5] R. Palamuttam et al., "SciSpark: Applying in-memory distributed computing to weather event detection and tracking," Big Data (Big Data), 2015 IEEE International Conference on, Santa Clara, CA, 2015, pp. 2020-2026.

[6] Sidhanta, Subhajit, Wojciech Golab, and Supratik Mukhopadhyay. "OptEx: A Deadline-Aware Cost Optimization Model for Spark." arXiv preprint arXiv:1603.07936 (2016).

[7] Kang, Hyejeong, et al. " A SLA-based VM Auto-Scaling Method in Hybrid Cloud Computing for Scientific Computational Applications." , Journal of KIISE : Computer Systems and Theory 40(6), 2013.12, pp. 266-273.

[8] Ahn, Yoonsun, et al. " An Auto-Scaling Technic of Virtual Resources on Hybrid Clouds for Scientific Applications.", Journal of KIISE : Computer Systems and Theory 41(4), 2014.8, pp. 158-165

[9] Gandhi, Anshul, et al. "Autoscaling for Hadoop Clusters." (2016).

[10] Jacob Leverich and Christos Kozyrakis, “On the energy (in)efficiency of Hadoop clusters, ACM SIGOPS Operating Systems Review, Vol. 44, No. 1, pp. 61-65, 2010

[11] Xueying Wang, Zhihui Lu, et al. “InSTechAH: An Autoscaling Scheme for Hadoop in the Private Cloud”, Services Computing (SCC), 2015 IEEE International Conference on, pp. 395-402, 2015.

[12] OpenStack, https://www.openstack.org/