Auto-scaling of scientific workflows in Kubernetes

Bartosz Baliś[1][0000−0002−3082−4209], Andrzej Broński[1], and Mateusz Szarek[1]

[1]AGH University of Science and Technology,
Institute of Computer Science, Krakow, Poland
balis@agh.edu.pl

Abstract. Kubernetes has gained extreme popularity as a cloud-native platform for distributed applications. However, scientific computations which typically consist of a large number of jobs – such as scientific workflows – are not typical workloads for which Kubernetes was designed. In this paper, we investigate the problem of *autoscaling*, i.e. adjusting the computing infrastructure to the current resource demands. We propose a solution for auto-scaling that takes advantage of the known workflow structure to improve scaling decisions by predicting resource demands for the near future. Such a *predictive autoscaling policy* is experimentally evaluated and compared to a regular *reactive policy* where only the current demand is taken into account. The experimental evaluation is done using the HyperFlow workflow management systems running five simultaneous instances of the Montage workflow on a Kubernetes cluster deployed in the Google Cloud Platform. The results indicate that the predictive policy allows achieving better elasticity and execution time, while reducing monetary cost.

Keywords: scientific workflows, auto-scaling, Kubernetes

1 Introduction

Kubernetes is a container orchestration system which has gained extreme popularity as a universal platform for management of complex distributed applications. However, scientific computations, in particular scientific workflows which are large graphs of tasks [7] – are not typical workloads for which Kubernetes was designed. We propose and evaluate a solution for auto-scaling of Kubernetes clusters tailored to scientific workflows. The main contributions of this paper are as follows: (1) two auto-scaling policies specific for scientific workflows and Kubernetes – reactive and predictive – are proposed and implemented. (2) The policies are experimentally evaluated and compared to a standard Cluster Autoscaler, using the HyperFlow Workflow Management System [1] running a workload of multiple large scientific workflows on a Kubernees cluster consisting of 12 nodes with 96 cores.

The paper is organized as follows. Section 2 presents related work. Section 3 presents the proposed solution for predictive autoscaling. Section 4 contains experimental evaluation of the solution. Section 5 concludes the paper.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_5

2 Related Work

The basic method of scaling is *reactive* [2], wherein a scaling manager adjusts resource allocation to their actual use, based on such metrics as the number of requests per minute, or the number of active users.

Several autoscaling approaches for scientific workflows have been proposed in the context of cloud infrastructures, e.g. using AWS Spot instances [9]. Cushing et al [3] introduce a scaling policy that relies on prediction of task execution times and estimates future demand based on the currently queued tasks. Versluis and others [10] compare several scaling policies using trace-based simulation.

Unlike the autoscaling policies for general applications, specific policies can be applied to graphs of tasks. In [6], two such policies – *Plan* and *Token* are proposed. The first one makes predictions and partial analysis of execution – so it requires knowledge of the graph structure and estimates for individual tasks. The second policy only uses information about the structure of the graph to estimate its *Level of Parallelism*. Although the quality of the estimation in the *Token* policy strongly depends on the structure of the graph, the paper [4] shows that it brings significant results for popular computational tasks. The work [5] presents an autoscaler of the *Performance-Feedback* type based on the *Token* policy, which supports many simultaneously running *workflows*, and also shows the integration architecture with *Apache Airflow*.

In summary, no existing work investigated auto-scaling of scientific workflows specifically in the context of Kubernetes using in-situ experimental evaluation.

3 Auto-scaling scientific workflows in Kubernetes

We adopt a solution wherein the autoscaling process can be viewed as a *MAPE* loop [8] consisting of four steps: (1) *Monitoring* – collecting information about the state of the cluster and the workflow execution state; (2) *Analysis* – predicting the future execution state and resource demands; (3) *Planning* – finding the best scaling action that accommodates the predicted workload; (4) *Execution* – performing the *scaling action*.

3.1 Monitoring

The basis of the autoscaler operation is the awareness of the current state of the cluster and the computations. We use the *SDK API* to retrieve information about *Pods* and *Nodes*, where the former determine the current demand for resources, and the latter their current supply. The demand for resources is calculated based on the *resource requests* of the Kubernetes Pods that run workflow tasks, so as to match the algorithm used by the Kubernetes scheduler. The supply of resources is the total amount of CPU and memory available on the worker nodes, reduced by the resources reserved by the Kubernetes components (e.g. Kubelet).

The workflow execution status is tracked through the events emitted by the HyperFlow workflow management system which we use to experimentally evaluate the autoscaling policies.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_5

3.2 Analysis

The purpose of this step is to determine the demand for resources in the near future. To this end, a given time period, e.g. 5 minutes, is divided into smaller time frames, e.g. seconds, and each frame is assigned a specific amount of resources. To estimate how much resources are needed in the upcoming time frames, we **simulate the further execution of currently running workflow graphs** using the method described in [6], so as to predict **which tasks will be running in parallel in a given time frame**. An illustration of the analysis process is presented in Fig. 1.

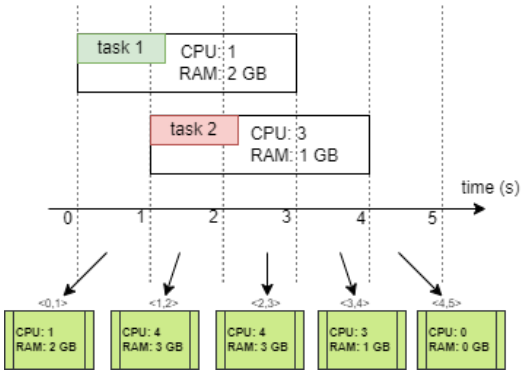


Fig. 1: Analysis of resource requests in time frames.

To calculate the demand for resources, we use the CPU and memory requests of the containers, in a similar way they are used by the Kubernetes scheduler. To take into account both CPU and memory requirements at the same time, we need to combine these two measures that have completely different units. To this end, both CPU and memory utilization are expressed as a percentage of the respective available resources and then added up.

3.3 Planning

The purpose of the planning step is to determine which action will be the most beneficial: scaling up or down by a certain number of machines, or perhaps no scaling. The outcomes of all possible decisions are checked within a given time limit, at specified time intervals (e.g. 5-frame sampling). The number of all combinations – decisions about **how to scale** and **when** – is the product of the maximum number of machines and the number of samples within the time limit. This number is sufficiently small so that all combinations can be checked.

Each combination is assigned a value of S (*score*), which represents how far the scenario deviates from an optimal match of resource demands and supplies. The deviation from the optimal resource allocation – either in the form of

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_5

underprovisioning or overprovisioning – of resource *res* is given by formula 1. There, *supply* is the value of the supply of resource *res* in the cluster taking into account the scaling action, and *demand* is the demand for resource *res*.

$$D_{res} = |\frac{demand_{res} - supply_{res}}{demand_{res}}| \tag{1}$$

Additionally, in frames where one of the resources is over- and the other is underprovisioned, the value of D is set to 0, so as to optimize runtime while accepting an additional cost.

The score S is then calculated, using formula 2, as a mean value of all resource over- and undersupplies over n time frames.

$$S = \frac{1}{n} \cdot \sum_{i=0}^{n} (\frac{U_{mem_i} + U_{cpu_i}}{2} + \frac{O_{mem_i} + O_{cpu_i}}{2}) \tag{2}$$

For each configuration (scaling decision), we also estimate its resulting **monetary cost**. Whenever two configurations are equal in terms of score, we choose one with a lower cost.

3.4 Execution

The final step is performing the scaling action, and this is done via the API of a given cloud provider. Due to the fact that scaling may take several minutes, the execution is asynchronous, i.e. we do not wait for information about the completion of scaling. To eliminate too frequent scaling attempts, after a scaling action we impose a *scaling cooldown period*, e.g. 2 minutes, during which no scaling actions are allowed.

3.5 Autoscaling policies – reactive vs. predictive

In order to evaluate the impact of leveraging the knowledge of the workflow structure on the quality of scaling, we distinguish two *autscaling policies*. In the ***reactive policy***, we assume that the future demand for resources is identical to the current one, so the planning phase simply adapts to the current situation. With the ***predictive policy***, on the other hand, knowledge of the workflow structure is used to estimate future demand, as described in section 3.2.

4 Evaluation

To evaluate the proposed predictive autoscaling algorithm, we experimentally run the same workload using three different configurations: (1) reactive policy, (2) predictive policy, (3) standard Cluster Autoscaler (which uses its own implementation of a reactive policy).

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_5

4.1 Experiment setup

To run the experiments, we used the Google Cloud Platform (GCP), with a Kubernetes cluster consisting of one master node to run the HyperFlow components, and a pool of worker nodes, using the *n1-highcpu-8* machine type, scalable from 1 to 12 nodes (up. to 96 cores).

Let us note that when the scaling action is performed, *Kubernetes* may try to stop a *Pod* and start it on another machine. In line with our assumptions, we care first about time and secondly about the budget. Thus, we define *PodDisruptionBudget* for all tasks, which consequently blocks the removal of the machine until all previously started tasks are finished.

The test workload was the Montage (degree 2.0). The experimental workload consisted of 5 instances of this workflow running simultanoeusly. The details are shown in Table 1. The CPU requests were set to 0.5 for mDiffFit and mBackground tasks and to 1.0 for all the others. The memory request was set to 256 MiB for all tasks.

Task type (agglomeration)	Count	Task type (agglomeration)	Count
mProject (3x / 3s)	1535	mImgtbl	5
mDiffFit (12x / 6s)	4310	mAdd	5
mConcatFit	5	mShrink	5
mBgModel	5	mJPEG	5
mBackground (12x / 4s)	1535		

Table 1: Experimental workload – 5 instances of the Montage workflow.

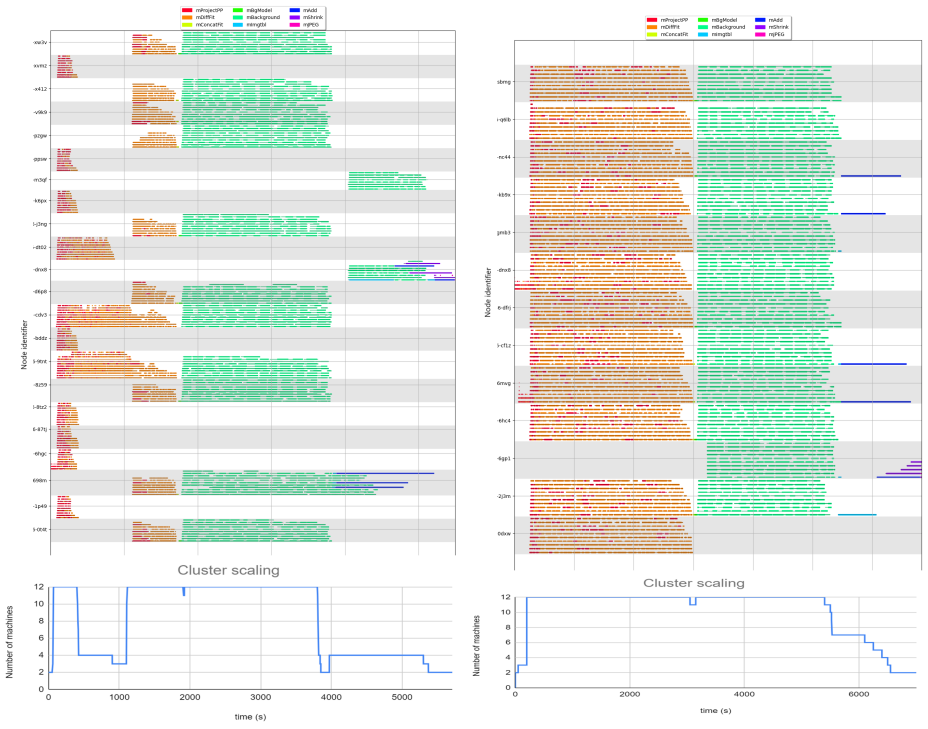
HyperFlow supports agglomeration (clustering) of tasks to reduce the overhead of starting excessively many Pods. In the case of Montage, tasks for the three parallel stages – especially *mDiffFit* and *mBackground* – are rather short, so that Pod creation time (typically about 2s) can introduce a significant overhead. Configuration of task agglomeration is also shown in Table 1. For example, *12x / 6s* means that HyperFlow will submit the tasks of a given type in batches of 12, but the maximum wait time to form a batch is 6 seconds.

4.2 Results

Fig. 2 shows the visualization of the execution traces of the experimental workload, along with cluster scaling, for the React and Predict policies, respectively. Because of space limitations, a similar visualization for the CA-based execution is not shown. However, Table 2 summarizes the key metrics for all three cases (CA, React and Predict): total execution time and the cost of execution.

It can be seen that the two executions of the experimental workload, respectively driven by the React and Predict policies, are quite different. The Predict policy results in more scaling decisions overall. The visualization of the execution trace for the Predict policy looks less 'compact' but this is only due to the fact that in total 22 different nodes are involved in the execution with the Predict policy, compared to 13 nodes for the React policy.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_5



(a) Predict policy.

(b) React policy.

Fig. 2: Execution of the experimental workload.

Montage-Degree 2.0	*react(CA)*	*predict*	*react*
Execution time [s]	6000	5709	6837
Cost [$]	7.05	4.53	6.95

Table 2: Experiment results summary for different auto-scaling policies.

The results indicate that the Predict policy performed significantly better than both CA and React ones. Not only the achieved execution time was the shortest (better by 5% compared to the second best CA policy), but it resulted in the lowest cost of execution ($4.53 compared to $\sim$ $7 for the two other policies). It is reasonable to conclude that with the execution time of about 100 minutes, the overhead of starting and shutting down more nodes was not significant.

Another interesting observation is that our implementation of the reactive policy performed significantly worse than the policy of the standard Cluster Autoscaler. While this is not the key result of the experiment, the reasons for this will be the subject of future investigation.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_5

5 Conclusions and Future Work

Autoscaling enables elastic resource allocation which meets the current demands, avoiding undesirable underprovisioning (hurting performance) or overprovisioning (increasing cost) of resources. We presented an autoscaling solution for running scientific workflows in a Kubernetes cluster. The proposed autoscaling policy was designed to take advantage of the knowledge of the workflow structure in order to predict the resource demands in the near future and thus, hypothetically, achieve better scaling decisions than a reactive policy. This hypothesis was confirmed experimentally.

Future work involves further experiments with different types of workflows and investigation of the impact of various factors on the execution and autoscaling, such as fine-tuning of resource demands of workflow tasks.

Acknowledgements. The research presented in this paper was partially supported by the funds of Polish Ministry of Education and Science assigned to AGH University of Science and Technology.

References

1. Balis, B.: Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows. Future Generation Computer Systems 55, 147 – 162 (2016)
2. Chieu, T.C., Mohindra, A., Karve, A.A., Segal, A.: Dynamic scaling of web applications in a virtualized cloud computing environment. In: 2009 IEEE International Conference on e-Business Engineering. IEEE (2009)
3. Cushing, R., Koulouzis, S., Belloum, A.S., Bubak, M.: Prediction-based auto-scaling of scientific workflows. In: Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science. pp. 1–6 (2011)
4. Ilyushkin, A., Ghit, B., Epema, D.: Scheduling workloads of workflows with unknown task runtimes. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 606–616 (2015)
5. Ilyushkin, A., Bauer, A., Papadopoulos, A.V., Deelman, E., Iosup, A.: Performance-feedback autoscaling with budget constraints for cloud-based workloads of workflows. CoRR abs/1905.10270 (2019), http://arxiv.org/abs/1905.10270
6. Ilyushkin, A., et al.: An experimental performance evaluation of autoscaling policies for complex workflows. p. 75–86. ICPE '17, ACM (2017)
7. Juve, G., Chervenak, A., Deelman, E., Bharathi, S., Mehta, G., Vahi, K.: Characterizing and profiling scientific workflows. Future generation computer systems 29(3), 682–692 (2013)
8. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A review of auto-scaling techniques for elastic applications in cloud environments. Journal of Grid Computing 12(4), 559–592 (12 2014)
9. Monge, D.A., Garí, Y., Mateos, C., Garino, C.G.: Autoscaling scientific workflows on the cloud by combining on-demand and spot instances. Computer Systems Science and Engineering 32(4), 291–306 (2017)
10. Versluis, L., Neacsu, M., Iosup, A.: A trace-based performance study of autoscaling workloads of workflows in datacenters. In: 2018 18th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 223–232. IEEE (2018)

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_5