Self-configuration of the Number of Concurrently Running MapReduce Jobs in a Hadoop Cluster

Bo Zhang, Filip Křikava, Romain Rouvoy, Lionel Seinturier
University of Lille 1 / INRIA Lille, France
first.last@inria.fr

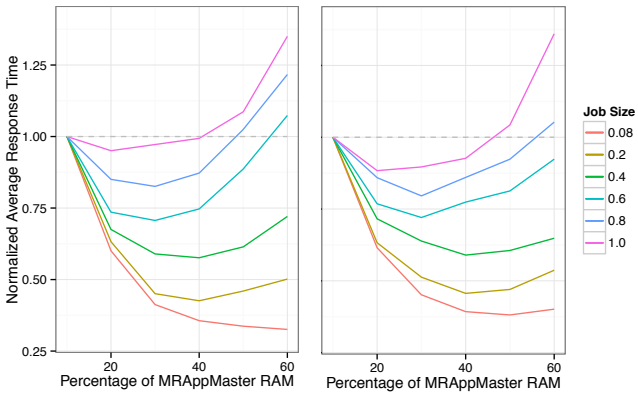
***Abstract*—There is a trade-off between the number of concurrently running MapReduce jobs and their corresponding map and reduce tasks within a node in a Hadoop cluster. Leaving this trade-off statically configured to a single value can significantly reduce job response times leaving only suboptimal resource usage. To overcome this problem, we propose a feedback control loop based approach that dynamically adjusts the Hadoop resource manager configuration based on the current state of the cluster. The preliminary assessment based on workloads synthesized from real-world traces shows that the system performance can be improved by about 30% compared to default Hadoop setup.**

I. Introduction

Hadoop is a popular implementation of the MapReduce (*MR*) programming model for large-scale data analysis. Despite its wide adoption, its out of box performance is often far from ideal leaving only suboptimal resource usage. Optimizing its performance remains difficult due to Hadoop high customization resulting in search space covering hundreds of configuration parameters. Recently, many researchers have been focusing on optimizing some of these parameters primarily working with job-level [1], [2], [3] or cluster-level configuration [4], [5]. In this work, we focus on configuration of YARN, the Hadoop cluster resource-management component. We propose an approach that dynamically adjusts the ratio between the number of concurrently running MapReduce jobs and their corresponding map an reduce tasks within a cluster node.

YARN separates the responsibilities between running Hadoop MR jobs[1] and their concrete map and reduce tasks into individual entities: a global, cluster-wide `ResourceManager`, a per node slave `NodeManager`, a per job MR application master (running within a `NodeManager`)—*i.e.* `MRAppMaster`, and a per MR task container (also running within a `NodeManager`)—*i.e.* `YarnChild`. Each `NodeManager` therefore runs both MR jobs (`MRAppMaster`) spawned by `ResourceManager` and a number of map and reduce task containers (`YarnChild`) that were spawned by the running `MRAppMasters`.

The ratio between the number of concurrently running MR jobs and MR tasks can significantly affect the overall jobs response time as is shown in Figure 1. The plot shows a non-monotone behavior between the response time and the number of running `MRAppMasters` represented as the maximum amount of node memory available for MR job masters. The rest of the free memory is allocated to the individual job map and reduce tasks. This phenomenon is caused by two problems: *under-allocation* and *over-allocation* for `MRAppMaster`. The former limits the number of MR jobs that can run in parallel, resulting in `NodeManager` underutilization by not having enough MR tasks to run. The latter, on the other hand, allocates too much memory for `MRAppMaster` which leads to insufficient memory to process the individual MR tasks.



(a) 100 requests/peak (b) 50 requests/peak

Figure 1. Relationship between the job size, the percentage of RAM for `MRAppMasters`, and the average job response time (testbed details in Section III).

Moreover, in the case of time-varying workloads, a statically configured threshold negatively impacts the performance as it trades one type of MR jobs (one job size) for another. A MR job size is the ratio between the maximum concurrent resource requirements of the job and the total amount of resources provided by the Hadoop cluster.

II. Approach

The objective of this work is to prevent both under- and over-allocation of the MR application masters in a Hadoop cluster. The `ResourceManager` capacity scheduler[2] has a configuration parameter that controls the maximum concurrently running applications[3] (C).

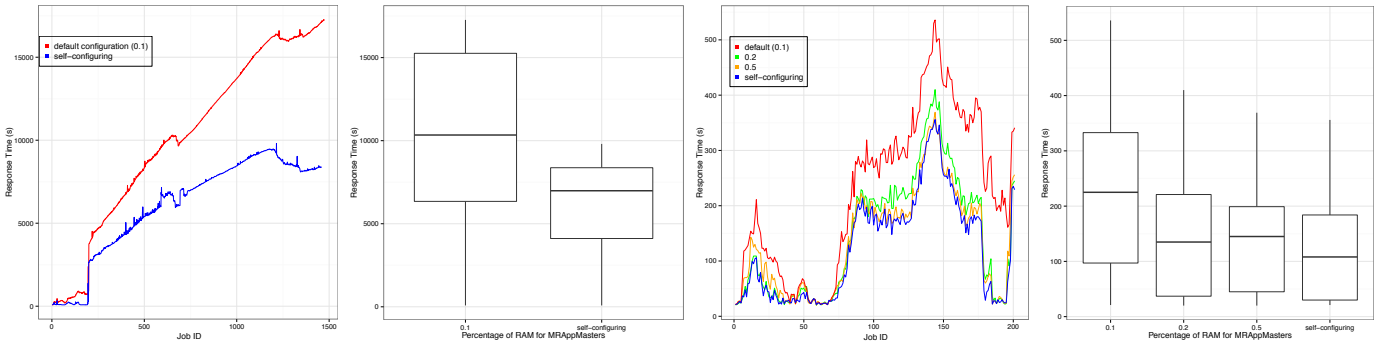
Our approach is based on a basic closed feedback control loop that consists of *monitoring*, *controlling* and *reconfiguration* components:

- *Monitor* periodically (every t seconds) measures the amount of memory used by both `MRAppMaster` application masters (M^{AM}) and `YarnChild` task containers (M^{YC}), as well as the number of idle MR jobs waiting to be scheduled

[1] A distributed application in general since YARN is not limited to just MR.

[2] The default scheduler for Hadoop 2.x.

[3] `yarn.scheduler.capacity.maximum-am-resource-percent`



(a) W1: job response time (b) W1: job response time distribution (c) W2: job response time (d) W2: job response time distribution

Figure 2. Preliminary results

by the `ResourceManager` (n_{idle}). The memory usage is gathered from each cluster node i together with the total available memory M^T and the results are summed up.

- *Controller* is responsible to derive a new value for the scheduler configuration parameter. At this point we use a simple algorithm that incrementally increases or decreases the value depending whether the system is in under- or over-allocation case respectively. It works as follows:
  (1) if the normalized amount of memory ($\frac{\sum M_i^{AM} + \sum M_i^{YC}}{\sum M_i^T}$) is less then a given threshold T_1 and there the number of idle jobs is increasing (n_{idle} at t_n is less then n_{idle} at t_{n+1}), the system is *under-allocated* and thus C will be incremented by a single step s that defaults to 0.05.
  (2) if the system is not under-allocated and the normalized amount of memory used by `YarnChild` is ($\frac{\sum M_i^{YC}}{\sum M_i^T}$) is less then a given threshold T_2, the system is *over-allocated* and thus C will be reduced by the same single step s.
- *Reconfigure* component simply modifies YARN setting with the new value. It does that by changing the capacity scheduler configuration file and issuing a command that makes YARN reload its settings.

III. Preliminary Results

A preliminary assessment of our work has been done on two experiments that use two different MR workloads. Both workloads were generated by the SWIM, a tool that generates representative test workloads by sampling historical MapReduce cluster traces from Facebook [6]. In the first experiment, the workload consisted of 1450 simple MR jobs of one map and one reduce task. In the second experiment, the workload contained 200 jobs with the size ranging from 2 to 8 tasks (1 map / 1 reduce to 7 maps / 1 reduce). The testbed used to evaluate our approach was a cluster of 5 hosts each with 7GB RAM, 2x4 cores Intel CPU 2.83GHz. The experiments results are shown in Figure 2.

In Figure 2a the two lines represent the variations of MR jobs response time in the first experiment. The red line captures the behavior of a vanilla Hadoop configuration (*i.e.* $C = 0.1$) while the blue line shows the behavior using our approach. It shows that even if the submission rate keeps increasing (*e.g.* a load peak around the request Id 230), the response time will be reduced by our approach. Furthermore, after the load peak, the difference between the two lines widens and is even more apparent since our approach lets more MR jobs be run in parallel. In summary, the average response time of the self-configuring C is reduced about 30% compared to the vanilla configuration (*cf.* Figure 2b).

In Figure 2c we show the results of the second experiment. This time we compare our approach not only to a vanilla Hadoop configuration, but we make several runs for different values of $C \in \{0.1, 0.2, 0.5\}$. Again, we can observe, that our approach performs much better than a vanilla Hadoop configuration. With increasing C the response time decreases in most cases, but it still remains statically set and thus does not respond to to the workload dynamics. The self-configuring approach, however, keeps changing the settings and thus adjusts better to the varying runtime condition.

IV. Conclusion and Further Work

Leaving the trade-off between the number of concurrently running MR jobs and their corresponding map can significantly reduce job response times. In this paper, we have proposed a simple feedback control loop that dynamically adjusts the YARN configuration in response to the state of the Hadoop cluster. The current work in progress is in developing a proper control-theory based controller and conducting a full evaluation on larger experiments.

Acknowledgments: This work is partially supported by the Datalyse project www.datalyse.fr.

References

[1] H. Herodotou, H. Lim, G. Luo, and N. Borisov, "Starfish: A self-tuning system for big data analytics." *Conference on Innovative Data Systems Research*, 2011.

[2] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based auto-tuning of mapreduce," in *Euro-Par 2013 Parallel Processing*, 2013.

[3] C. L. et al., "An adaptive auto-configuration tool for hadoop," in *International Conference on Engineering of Complex Computer Systems*, 2014.

[4] P. Lama and X. Zhou, "Aroma: automated resource allocation and configuration of mapreduce environment in the cloud," in *International Conference on Autonomic Computing*, 2012.

[5] K. Chen, J. Powers, S. Guo, and F. Tian, "Cresp: Towards optimal resource provisioning for mapreduce computing in public clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, 2014.

[6] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, "The case for evaluating mapreduce performance using workload suites," in *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011.