Software Defined Concurrency Control in Hadoop

Rohan Naik

***Abstract*— Hadoop is commonly used as a open source framework for distributed data analysis on a large scale. The architecture of Hadoop is such that it deploys fixed sized containers for MapReduce job processes. The default Hadoop scheduler schedules the job and manages the workload on the basis of statically configured resource allocation settings. This project demonstrates an implementation that dynamically reconfigures these resource allocation settings in runtime through software definition. The said implementation is built using python and shell scripts over Hadoop framework. Essentially, this implementation provides two control knobs for dynamic tuning and concurrency control of MapReduce jobs at runtime. These knobs aim to reduce the runtime/ makespan of MapReduce jobs as compared to when they are run with static configuration.**

I. INTRODUCTION

Hadoop is an open source frame work that is widely used for data-analytics that involves processing of large datasets. This framework allows the user to configure cluster in master-slave setting and MapReduce programs can then be deployed on the cluster. YARN (Yet Another Resource Negotiator) is a data operating system introduced for hadoop 2 and the architecture follows a container based approach with several components as explained with Figure1 in description below:

For Master Node:

Resource Manager (RM):

It consists of *Application Manager* that acts as a Resource Negotiator to negotiate the first container from Resource Manager to start a new job. It is just a management node

Scheduler

(capacity-scheduler, fair-scheduler or customized)

Allocates resources to running applications after checking the constraints, capacities etc once the application is started.

For Slave Nodes:

Node Manager: It is responsible to launch containers and monitor their usage in terms of CPU, Memory, IO etc.

Application Master: It negotiates with Resource Manager to schedule tasks and tracks their state.

YarnChild: It is the worker unit of Hadoop that runs the mapper or reducer computations and

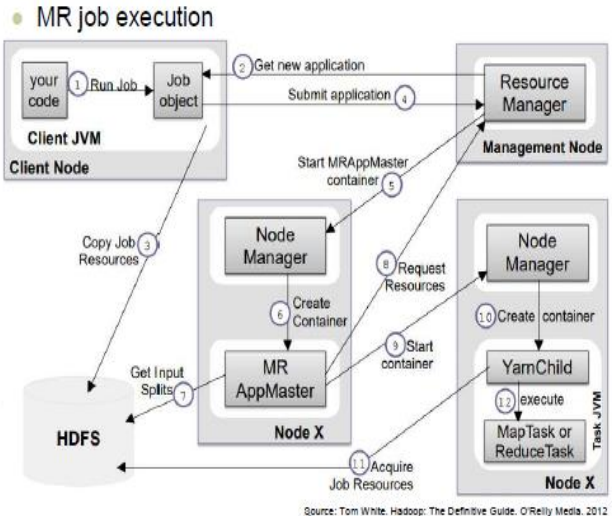

Figure 1: Hadoop 2 workflow for MR jobs
(Courtesy: Lecture Slides EEL6871 – Dr. Jose Fortes)

The main challenges involved in performance optimization of such analytics frameworks are mentioned below, all of which lead to poor performance and underutilization of resources:

1. Diversity of analytics jobs and frameworks
   → Hard to predict job performance
2. Dynamics of jobs resource demands
   → Challenging dynamic resource management
3. Variety in number of configuration parameters → Hard to tune manually

The cluster is statically configured by Hadoop properties in form of xml files. However, these settings are static and once configured, we are stuck to those settings. The MR jobs are diverse in nature and have different resource demands (IO, CPU, Memory etc). Therefore, the cluster is almost always underutilized and works best only for a certain narrow set of applications.

With the aforementioned background, we define a control system with two input variables. Namely, MaxJobs and

MaxContainers. The MaxJobs controls the total number of MR jobs that can be taken up by Scheduler at the global level in master nide. And the MaxContainers input which varies the slave nodes. controller that aims at maximum resource utilization of cluster resources.

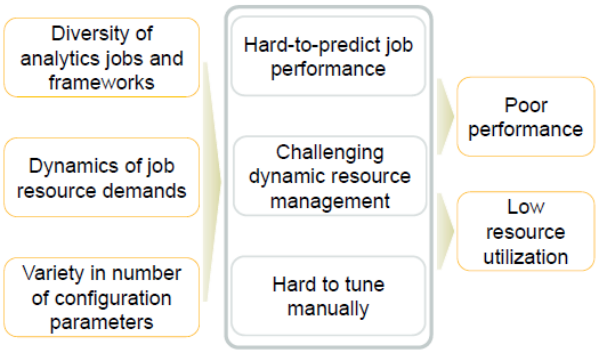

Figure 2 Challenges in performance optimization for data analytics framework
(Courtesy: Lecture Slides EEL6871 – Dr. Jose Fortes)

Together, these input variables form a global controller for master node and a local controller for each slave node. The resulting design is a software defined feedback

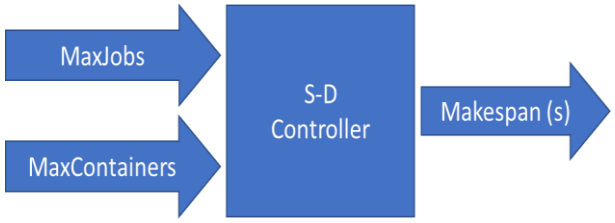

Figure 3 Controller where
y = makespan and u1, u2 = Maxjobs, Maxcontainers

II. Implementation Details

Hadoop by default, uses capacity-scheduler for scheduling its MR jobs. As a part of the optimization algorithm, we used a property of the capacity scheduler called AMRP: Application Max Resource Percentage, which is a value between 0 and 1 and is expressed as a fraction of total number of containers allowed for running Application Master.Thus varying this value controls the number of applications allowed to run on the Hadoop cluster for a given number of cluster.

A. Capacity Scheduler:

The specific properties and their preset values are defined in the capacity-scheduler.xml configuration file in Hadoop directory. The general functionality of capacity scheduler is as described below:

Capacity scheduler forms queues instead of pools for tasks and has resources allocated to each of these queues (typically 1 queue = 1 client). The resource utilization of each of these queues is monitored and excess of unutilized resources are allocated to the high priority application from other queues. i.e. capacity of the queues is shared among other queues based on application priority and resource availability.

Example. If there are 5 applications using a cluster, they will have 5 queues. If each of them have a task running, they shall use the minimum guaranteed resources, that is 20% of the cluster. If there are just four of the applications using the cluster and queue of application#5 is empty, it will be distributed among other applications based on priority or pricing offered. Assuming, all have the same priority, all will share 25% of the resources now. Similarly for 3 empty queues of C, D,E applications, A and B will share 50% resources each.

B. *Container based functionality*

Hadoop allocates computing resources in the form of resource group units known as Hadoop Containers. The configuration of a container is given in yarn-site.xml file in Hadoop directory.
Whenever the application in process requires more resources or if a new application is to be started, the Resource Manager deploys one or more containers based on requirement. In this project, the configuration file uses containers with 1 GB memory and 1 virtual CPU core.

C. *Controller Architecture and Control Mechanism [2]*

The architecture of the controller follows a MAPE model and has a Hierarchical control of resource allocation on master and slave. Therefore, the resulting architecture has two controllers. A local controller that controls MaxContainers (YarnChild) in each slave separately, and a global container that controls the MaxJobs if the MaxContainers setting is not able to fully utilize the resources. Figure 4 shows the controller architecture.

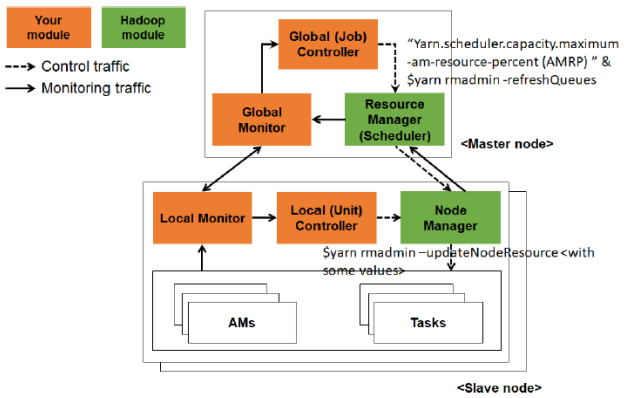

Figure 4 Controller Architecture

This architecture is realized with the help of following components:

MaxContainers: Maximum number of containers (computational units) allowed on a slave node
MaxJobs: Maximum number of Jobs allowed to run on cluster. Indirectly set by setting AMRP value in capacity-scheduler.xml

1. Local Monitor: It gathers the Node and MRJob related information every second and writes it to a log file. Following information is collected:

NodeInfo: Timestamp, CPU usage (user+nice+system+steal), CPU (iowait), CPU (idle), number of context switches per second (CTSW), transactions per second (TPS), total memory of a node in KB, used memory in KB, used memory (%), size of the running queue, number of blocked tasks, max. device utilization, max. network interface utilization

TaskInfo: Timestamp, CPU usage (%) by MR tasks, memory usage (%) by MR tasks, number of containers used by tasks, number of AMs, number of YarnChild

These statistics are collected at each slave node. They are used by the local monitor to change the MaxContainers as well are periodically sent to Global Monitor to calculate aggregate usage of resources.

2. Local Controller:

Local Controller monitors the node's CPU usage and decides whether the MaxContainers are sufficient or not.

The controller checks:

If NodeCPU < Lower Threshold:
MaxContainer = MaxContainer + 1
IncrementFlag = True
Else if NodeCPU > Upper Threshold:
MaxContainer = MaxContainer -1
DecrementFlag = True
Else: Steady State Region → Do Nothing

Studies were carried out by using instantaneous utilization for NodeCPU as well as a moving average filter with average of last n sample. For larger and but intermittent workloads, moving average filter works more efficiently and gives a smooth transition. For smaller workloads the effect could not be studied well.

We used Hadoop Yarn APIs to implement these changes. For instance, local controller uses:

$yarn rmadmin -updateNodeResource <NodeID> <Memory> <vcores>

To update MaxContainers and

$yarn node -list

To obtain NodeID and Number of Container Information

3. Global Monitor:

The Global Monitor collects resource utilization information from Slave Nodes periodically (every 10 seconds for this project) and computes the aggregate resource utilization [2].

4. Global Controller:

The global controller keeps the aggregate resource utilization of the entire cluster in check. Its main task is to update correct value of AMRP. The controller changes the AMRP in two situations. 1. When local controller updates the MaxContainers but the ratio of AppMasters to MaxContainers needs to be constant to maintain constant MaxJobs.

2. When the aggregate resource utilization (%CPU in our case) remains underutilized (below 70%) or goes beyond 90% it changes the MaxJobs to accommodate more/ less number of jobs. Controller logic was same as that provided in local controller.

We used ElementTree API to change the value of AMRP in capacity-scheduler.xml [6]

After every update in xml file, we used the command refreshQueues to update the value in Hadoop Daemon[2]

In context of a MAPE controller design, following analogies can be established:

Local and Global Monitors, logfiles: **Monitoring**

Local and Global Controllers: **Analyze and Plan**

Yarn and ETree APIs : **Execute**

III. Experimental Setup

To carry out this study, Cloudlab was used as a platform to host the Hadoop cluster with 3 nodes. 1 master and 2 slaves. We used the Utah Infrastructure wherein each node had 16 physical CPU cores with each core having 4 GB memory.

Property settings and Threshold values:

- ✓ Slaves: 2
- ✓ Container Size: 1024
- ✓ Initialization value of slaves: 32 GB, 32 Containers Allowed
- ✓ Initialization value of AMRP: 0.2
- ✓ Local Controller CPU usage:
  - Average>90, decrease containers
  - Average<70, increase containers
  - 70<Average<90 do nothing – all is well! (values require finetuning)
- ✓ Global Controller aggregate usage:
  - Total Resource >180, decrease maxjobs
  - Total Resource < 140, increase maxjobs
  - 70<TR<90 do nothing – all is well!

IV. Results

The controller was tested (PS: without establishing synchronization between master and slave) and following observations were noted for a customized workload comprising of 2 Grep and 2 WordCount problems running concurrently.

Config	Makespan Time in Seconds				
	Grep1	Grep2	Wordcount1	Wordcount2	Total(s)
Static (without controller)	61	63	41	43	208
	52	47	31	36	166
	Average				**187**
With Controller	38	40	23	22	123
	38	38	28	23	127
	42	41	20	24	127
	Average				**126**

Table 1: Makespan Results for customized workload

V. Areas of Improvement

A detailed study needs to be carried out to support this preliminary work so as to present more profound and accurate results.

We only took CPU usage as a deciding factor but some MR jobs are IO intensive as well. Therefore the controller design should include IOwait times, ContextSwitches, memory etc. into account for a holistic design.

A more realistic workload needs to be designed to test the implementation in all possible scenarios.

The moving average control algorithm needs to be studied under various workload and the size of moving window as well as the gain and other control parameters need to be determined.

Conclusion

As seen in table 1, the makespan does come down on using the controller, but the readings are not sufficient and repeatable. Also, the workload was light and of short duration, not realistic enough to test the implementation accurately.

Also, the synchronization between master and slave could not be completed due to unforeseen errors. As a result, it can be concluded that the dynamic reconfiguration, independent of the nature of job, is a feasible option to optimize Hadoop framework, but it requires careful study and analysis of the controller design.

Appendix

Screenshots attached in a separate file named appendix.

Acknowledgment

I would like to sincerely thank Dr. Jose Fortes for allowing me to work on this project that has given me profound insight into software defined control system and also enhanced my scripting skills. I would also like to thank the TA Giljae Lee for his endless support throughout the project.

References

[1] Lee, Gil Jae, and José AB Fortes. "Hadoop Performance Self-Tuning Using a Fuzzy-Prediction Approach." *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. IEEE, 2016.

[2] Lee, Gil Jae, and José AB Fortes. "Hierarchical Self-Tuning of Concurrency and Resource Units in Data-Analytics Frameworks." *Autonomic Computing (ICAC), 2017 IEEE International Conference on*. IEEE, 2017.

[3] Kc, Kamal, and Vincent W. Freeh. "Dynamically controlling node-level parallelism in Hadoop." Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on. IEEE, 2015.

[4] https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YarnCommands.html

[5] Vavilapalli, Vinod Kumar, et al. "Apache hadoop yarn: Yet another resource negotiator." Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.

[6] https://docs.python.org/2/library/xml.etree.elementtree.html