Intelligent scheduling on YARN

Yunfan Ye (yunfany)
Carnegie Mellon University

Xiaoguang Zhu (xiaoguaz)
Carnegie Mellon University

Abstract

In this project, we implement a Hadoop YARN scheduler, which communicates with YARN resource manager through Thrift. Besides, we build a YARN cluster that has heterogeneous resources on five physical machines provided by Emulab, on which we evaluate our scheduler. Specifically, we implement four basic scheduling policies: strict First-Come First-Served (s-FCFS), FCFS Random, FCFS Heterogeneous and Shortest Job First (SJF) Heterogeneous. Moreover, we optimize the scheduler under several constraints, i.e. soft and hard heterogeneous placement and the amount of utility. Finally, we illustrate other potential optimization and design a custom trace that may induce high variance between schedulers.

1 Introduction

Hadoop YARN allows different frameworks, like MapReduce, run on the same physical cluster [1], which attracts great attention from both academia and industry. On YARN, a good scheduling policy is always hot research topic and there are multiple criteria to optimize, including average completion time and utility.

To dive deeper into YARN scheduling, We first employ five bare metal machines on the remote Emulab cluster to virtualize a VM cluster, on which we build our custom version of Hadoop YARN. Second, we implemented four scheduling policies to minimize the average completion time. In the third step, we implement three heterogeneity-aware scheduling policies that greedily maximize utility.

2 Problem Settings

Here we have one resource manager (RM) and four node managers (NMs), and each NM has multiple virtual machines (VMs). Meanwhile we use thrift to send requests to our RM and RM will distributed them to different VMs.

For a scheduler, we should choose which job to run, when to run and which VMs to place this job on. However, in phase two, we should design four algorithms, i.e. first-come first-served (FCFS) Heterogeneous, shortest-job-first (SJF) Heterogeneous and greedy algorithm on utility. In phase three, we are required to greedily maximize utility. As a result, we do not have too much freedom and only have to make two decisions, i.e. we need to decide when to schedule the next job and which VMs should we place the jobs on.

Specifically, first, when we have enough resources, we can decide to schedule the job immediately. However, we could avoid making decisions immediately as well. One reason may be waiting for better resources. For instance, a job needs several VMs to complete. However, when the job finished, all VMs may not finish at exact the same time, and thus it makes sense to wait for several seconds, hoping to get better resources, e.g. machines on one rack. Another reason to delay scheduling is hoping for scheduling better future jobs. This could be possible in practice when we have enough history to mine on, yet, this is not good for us because we cannot predict the future in our scenario. Concretely, it is entirely possible that no better future jobs will ever come.

Second, the placement of job is of great concern, especially when the preferred resources are not satisfied. Take GPU jobs as an example. When preferred resources are not satisfied, we may decide to scatter the job among racks as much as possible, hoping to balance the load between racks. Or, we could also try to pack the job into on rack, hoping to leave one rack has a lot of free VMs, which can then serve as preferred resources for MPI jobs.

3 Scheduler Design

In this section, we will talk about the design of our scheduler in multiple aspects.

3.1 Data structures

We use a job queue to store all pending jobs. Specifically, for FCFS policy, the job must be executed in the order of their arrival. Therefore, if we have enough resources to fulfill the job, we schedule it immediately. However, for SJF policy, we automatically put the job into the queue as we might schedule them in any order. Also, we put the job into the queue if we do not have enough resources. Moreover, we use a status array to mark whether one particular is free or allocated.

3.2 Scheduling policy

We have three kinds of scheduling policy. Specifically, FCFS must schedule the job in the order of their arrival. SJF must first schedule the pending job with shortest running time. Finally, we introduce the concept of utility, which is defined as C - t_d - t_w, where C is a constant, t_d is the duration and t_w is the waiting time. The final policy is to greedily maximize the utility of every job. Regarding FCFS policy, we do not have any freedom to schedule the job, while for other two types of jobs, we can scan through the job queue, deciding when to schedule and where to place the job.

3.3 Heterogeneity aware placement

In our scenario, we have two kinds of jobs, GPU and MPI. GPU jobs prefer big machines while MPI jobs prefer machines in the same rack. Furthermore, our hardware environment is heterogeneous, i.e we have one rack of four big machines and three racks of six small machines. This constraint has a big effect on the decision process.

Besides, this constraint can be considered as soft or hard. Specifically, soft constraint means that we are free to schedule the job anywhere if the preferred resources are not satisfied. Conversely, the hard constraint means that we must schedule the job on the preferred resources. For soft constraints, we are allowed to run jobs even it cannot be scheduled to preferred resources. To be more clear, we can place them anywhere or make them remain pending if we failed to place them to preferred resources.

For hard constraints, we should decide which resources to be allocated to the next jobs. Here are our algorithms to find the preferred resources: for MPI job, our scheduler will first search the MPI racks, that is, we will first put MPI jobs on non-GPU machines. Then, if all non-GPU machine cannot fit, we will consider the GPU machines; for GPU job, we only check the GPU machines for the available resources.

Regarding soft constraints, we should design algorithm when the scheduler failed to allocate preferred locations. For MPI jobs, it means that we cannot allocate all VMs on one rack, so we will try to allocate the 4 highest machines (the non-GPU machines have higher number) to that job. For GPU jobs, we will try to allocate them to MPI racks.

4 Evaluation

To evaluate the effectiveness of our scheduler, we test two trace combinations with two policies, i.e. soft and hard, on one Emulab cluster with 5 physical machines. Specifically, we have one rack of four big machines and three racks of six small machines. Also, the two trace combinations are traceMPI-c2x6-rho0.70 and traceGPU-c2x6-rho0.70, which is named as 70, and traceMPI-c2x4-rho0.80 and traces/traceGPU-c2x4-rho0.80, which is named as 80.

To better illustrate, we present the results in Table 1. From the results, we found that soft policy works better than hard policy. Therefore, we choose soft as our final policy and test its effectiveness on the hidden trace. The result is also presented in Table 1.

Trace	Policy	Utility	E(T)
80	soft	45161.599	421.351
80	hard	41761.334	479.977
70	soft	28472.069	386.512
70	hard	26050.125	455.711
hidden	soft	42648.593	292.583

Table 1: Results table

5 Possible Optimization

In this section, we will talk about some optimization based on the evaluation of our result and the environment of our jobs.

5.1 Optimization with Constrain

Each GPU job needs two to four machines and prefers rack 0 with only four machines. Each MPI job need exactly four machines and prefers the machines in the same rack. What's more, the hazard of putting GPU jobs to unpreferred location is much bigger than doing the same thing to MPI. So all of these proprieties can be used in optimization.

5.1.1 Prevent combination of different type of machines

For one job, we may assign machines on different racks because we do not have preferred location. However, if we distribute both MPI machines and GPU machines to one job, there will be trouble on scheduling further jobs because we only have four GPU machines. Meanwhile, this situation happens only when there are not enough single type machines for that job, so even if we do not assign job to them, there won't be big waste on resources.

5.1.2 Prevent MPI jobs from GPU rack

As we mentioned, putting GPU job to unpreferred location has much more hazard than putting MPI job to unpreferred location. So we should arrange MPI job from occupying the GPU resources. That is, we put MPI job to MPI racks first and then consider GPU rack.However, if there are not MPI racks that have four machines, but GPU rack, we should go through and arrange the job into GPU rack because we cannot foresee the further jobs and it is the highest expectation of utility.

5.1.3 Delay the GPU job with three / four machines

One optimization idea may be adjusting the arrangement by the machines demanded by jobs. That is, if we receive a GPU job that needs more than two racks, we can arrange it to unpreferred locations because we can reserve more space for other GPU job. However, we think it is a bad idea, because we cannot predict the future.

5.2 Optimization without Constrain

Without the constrain on selecting jobs, we can do more optimization on this part.

Following the requirement of the writeup, we want to maximize the total utility. Here the utility u of a single job is

$$u = \begin{cases} 1200 - T & \text{if } T < 1200, \\ 0 & \text{others.} \end{cases} \tag{1}$$

where T equals to its completion time in second ($T = T_{waiting} + T_{running}$).

Meanwhile, the yarn will kill the job once it has waited more than 1200s. So we can assume that $T < 1200$ for every jobs. As the result, to maximize the total utility U, we can generate the following conclusion,

$$\begin{aligned} \max(U) &= \max(\sum_{i=0}^{n}(u_i)) \\ &= \max(1200 * n - \sum_{i=0}^{n}(T_i)) \\ &= \min(\sum_{i=0}^{n}(T_i)) \end{aligned} \tag{2}$$

As a result, to maximize the utility equals to minimize the total (average) completion time. Then the SJF in phase two should perform better than the greedy algorithms on utility.

Different from phase two, different (GPU) jobs use different numbers of machine. With more machines left we can arrange more jobs. So the utility of each machine is $u_{machine} = u/num_{machine}$, where $num_{machine}$ is the number of machines demanded by this job. To make it simple, we can modified the SJF algorithm. We can arrange the job with shortest running time on single machine, that is $\arg\min_i(T^{(i)}_{running}/num^{(i)}_{machine})$ This algorithm should work better than greedy algorithms on utility.

6 Custom Traces

In this section, we try to provide the most general trace to beat the too specific optimization and prevent schedulers from overfitting.

Assuming the GPU jobs and MPI jobs will com together, we design two trace files together.

At first we arrange four MPI jobs only. they should running at all the preferred location. However, one who do not assign it to preferred location may feel bad because we do not send further requests in first two minutes. Then comes the another MPI job with long execution times. One who do not assign job to GPU rack may assign this one to GPU rack, which is a disaster.

After that, we send three GPU jobs with the demanding of machine 2, 3, 2. When serving the second request, there will not be enough machines in GPU rack. We will run it in MPI machines so that we have enough machines for the third job.

At the 300 seconds, we should have 2 free GPU machines and 14 MPI machines. Then send three MPI tasks to fill it and then three GPU jobs. Most of people will assign both MPI machine and GPU machine to GPU job, but we will get free GPU machine soon, Here we will survive.

Then we design to beat the hard policy, so we send a bunch of MPI jobs first and then a bunch of GPU jobs. Finally we send a little general mixture of different jobs.

Acknowledgments

Thank 746 staff for designing this project and helping us get deeper understanding of YARN scheduling. Also, thank professors and Emulab for the physical resources.

References

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