Progress Based Container Scheduling Algorithm

Jahnavi Swetha P, Kaushik Sai A, Yuan Shen
Department of Computer Science
New York University
New York, NY 10012
jp5867@nyu.edu, ka2734@nyu.edu, ys4420@nyu.edu

Abstract

As more and more deep learning jobs are infiltrating our life, many people are choosing cloud providers for hardware demands to speedup their jobs. When a job arrives at the server, a manager have to choose from several workers to run the incoming job. The most common strategy of choosing workers takes the resource availability into account, e.g., CPU, memory, GPU. Those scheduling algorithms focus on the fairness between users and utilization of the servers. In this study we experimented with a progress based scheduling algorithm which aims to reduce the completion time and makespan of jobs. When scheduling the incoming containers, ProCon[1] not only considers the current resource availability but also accounts for the future resource utilization. We implemented ProCon algorithm on Google Kubernetes Engine[2] and compare the job completion time and makespan with the default scheduler of Kubernetes[3].Through the monitoring of all jobs, we observed that ProCon tries to balance the contention of resources on all the worker nodes and reduces the makespan and average job completion time. Our experiments has shown that ProCon improves the completion time by 37% and shows an improvement of makespan for up to 25%

1 Introduction

Whenever a job arrives at the cluster, the master node has to select a worker node to host the job. Selecting the most desirable worker for the incoming job is non-trivial and can be selected based on the constraints we have for the cluster. But, most of the scheduler algorithms that were proposed for Kubernetes try to select one node from the list of worker nodes based on the single criteria like the state of nodes such as the number of running containers or the amount of the resources utilized on the node and fail to consider the expected running time of the containers for progress based jobs that that are already running on the nodes. So, we would like to introduce an alternative scheduling algorithm for Kubernetes that considers multiple criteria along with the expected running time in the selection of the node for a newly submitted container which would benefit progress based jobs based on expected completion time and makespan based on multiple other criteria.

Assuming there are two worker nodes in the cluster. Two containers are running on the first worker node and one container is running on second worker node. Suppose the containers running on first worker node could be completed in next few seconds and the the container running on second node will take maximum time and take up the majority of the resources on the second worker node in the next seconds. The default scheduler tries to keep the incoming container on the second worker node to balance the work load and this will degrade the performance. This happens since the scheduler doesn't have an idea about the global state of the cluster infrastructure. If the scheduler is aware of the expected completion time of the tasks running on all the nodes, the incoming container could be placed on the second node.In this work, we create a Progress based container scheme which combines the resource usage with the expected completion time to select the worker node.

2 Background

Containers are the encapsulated applications with the necessary dependencies into a sandbox that provides platform independent run-time environment. Nowadays many cloud providers like Google, AWS, IBM etc., provides a way to deploy these containers on cloud. These containers can be of either short lived containers like batch jobs or long lived containers like logging and monitoring services. The cluster contains master node and worker nodes and the worker nodes are responsible for running these containers. While deploying the containers, we can specify the resources requirements and cluster configuration requirements of the container to the master node. The scheduling algorithm then selects a worker node based on the constraints to host the container.

The selection happens in two phases:
1. Filtering phase : Filter out the worker nodes that doesn't meet the requirements.
2. Scoring Phase : The scheduler then scores the remaining worker nodes based on the user specified placement scheme and selects the best one based on this score.

In this paper, we inspect the scoring phase and focus on the short lived containers. Usual placement schemes doesn't really consider the fact that short lived containers will release the used resources once they are completed. So, the estimated completion time plays an important role while scoring the working nodes. The estimated completion time depends on job itself, resource availability and the workload on the nodes. Despite not knowing many details about the jobs, we can still estimate the completion time of the jobs by constantly monitoring the progress rate of the short lived containers and the resources available on the node for their completion. Hence, this motivates the estimation of the completion time of the short lived containers.

3 Architecture

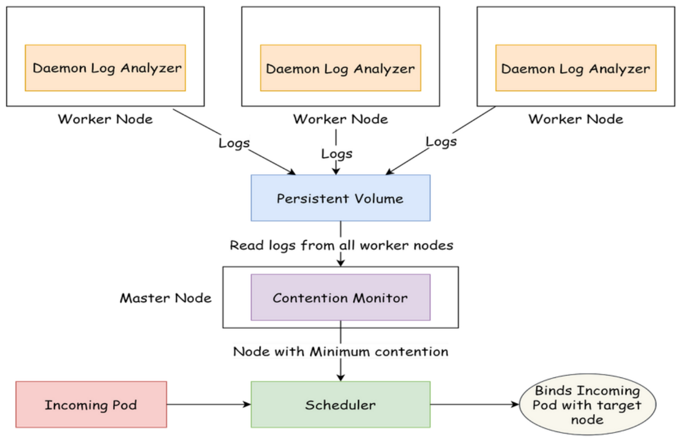


Figure 1: Architecture

In this section, we present the architecture of ProCon in detail
Log Analyzer: The log Analyst runs on each worker node of the kubernetes engine as a daemon and collects the logs from all the jobs running on that worker node and calculates the required parameters for the scheduling algorithm like contention rate of the worker node and places them on the persistent volume.

Contention Monitor : The contention monitor runs on the master node and has the global view of all the worker nodes. It calculates the resource availability of each worker node and the resource usage of each pod running on the worker nodes. It then reads all the logs presented by each worker node on the persistent volume and calculate the contention rate of each worker node based on the logs and the resource usage of the pods and the available resources on the worker node.

ProCon Scheduler: The scheduler collects data from the contention monitor and then picks the target worker node to host the incoming container based on the scores calculated.

4 Algorithm

CALCULATING ESTIMATED TIME OF COMPLETION:

The estimation of the completion time of the job depends on the job itself. The estimated time depends on the dataset, epochs and model, we cannot estimate the completion time without monitoring the logs. The batch jobs running on the worker reports their progress rate at regular intervals i.e., after every iteration. Since, ProCon is especially for short lived containers, we define the progress rate as the percentage of maximum number of epochs allowed that have been completed. For example, if the maximum epochs for the model is 100 and the current epoch is 5, then the progress rate if 5%.

We consider multiple jobs are running on the multiple worker nodes. We use J_{id} as the job id for each job and W_i represents the worker id. We calculate the unit progress rate time required for each job at a given time t by

$$f(J_{id}, t) = P(J_{id}, n) - P(J_{id}, n-1) \tag{1}$$

where $P(J_{id}, n)$ represents the timestamp at which Job J_{id} completed the latest epoch and $f(J_{id}, t)$ represents the time between last two epochs.

Algorithm 1 Calculate remaining time of Job

```
 1: procedure CALCULATE REMAINING TIME($J_{id}$)
 2:     System Initialization $J_{id}, W_i$
 3:     Parameters $P(J_{id}, n), CPU(J_{id}, t), MEM(J_{id}, t)$
 4:     for $J_{id} \in W_i$ do
 5:         $u(J_{id}, t) = \frac{0.8*CPU(J_{id},t)}{\sum_{J_{id}\in W_i} CPU(J_{id},t)} + \frac{0.2*MEM(J_{id},t)}{\sum_{J_{id}\in W_i} MEM(J_{id},t)}$
 6:         $i(J_{id}, t) = \frac{f(J_{id},t)}{u(J_{id},t)} * (MAX - n)$
 7:     end for
 8:     for $\forall J_{id} \in$W$_i$ do
 9:         $c(W_i) = Max(i(J_{id}, t))$
10:     end for
11: end procedure
```

The estimated completion time depends on the time required to complete the remaining epochs and the resources available for the job as all the jobs on the worker node shares the resources. So, we calculate the relative resource usage $u(J_{id}, t)$ of the Job J_{id} running on the worker $W_i$at time t.

$$u(J_{id}, t) = \frac{0.8 * CPU(J_{id}, t)}{\sum_{J_{id}\in W_i} CPU(J_{id}, t)} + \frac{0.2 * MEM(J_{id}, t)}{\sum_{J_{id}\in W_i} MEM(J_{id}, t)} \tag{2}$$

where $CPU(J_{id}, t)$ represents the current CPU utilisation and $MEM(J_{id}, t)$ represents the current memory utilisation of job J_{id} on worker node W_i at time t.

Since, the batch jobs are computationally intensive, we have given more weight to the CPU than memory. If the bottleneck resource is not CPU, then the weights can be modified accordingly. After combining the above equations, we calculate the remaining time indicator for the job J_{id} running on W_i at time t as

$$i(J_{id}, t) = \frac{f(J_{id}, t)}{u(J_{id}, t)} * (MAX - n) \tag{3}$$

where n is the current epoch number of the Job J_{id}. It then picks the maximum remaining time indicator among the jobs running on the worker node W_i. As the completion time also depends on the number of containers running on the worker node, when the contention rate is same for the worker nodes, we break the tie using the number of containers running on the selected target worker nodes and chose the one with the least number of containers.The Log Analyser daemon runs the Algorithm 1 by collecting the logs and computes the remaining time indicator for each job on the worker node it is running and calculates the

CONTAINER PLACEMENT:

The Contention monitor runs the Algorithm 2 to find the best possible target node. It removes the nodes that does not satisfy the requirements and collects the logs of the remaining nodes. It then chooses the one with the minimum contention rate and gives it to the scheduler. If it does not find the target node, it then waits and checks again to find the possible target node.

Algorithm 2 FIND THE TARGET NODE BASED ON CONTENTION RATES

```
1:  procedure CONTAINER PLACEMENT($W_i$)
2:      System Initialization $J_{id}, W_i, S = \{\}, TARGET = NULL$
3:      Parameters $P(J_{id}, n), c(W_i), Rem(R(W_i))$
4:      for $W_i \in W$ do
5:          if Remaining resources of $W_i >$ Required resources of incoming job then
6:              S.insert($W_i$)
7:          end if
8:      end for
9:      if |S| $= 0$ then
10:         Sleep 2s
11:         Jump to Line 4
12:     end if
13:     for $\forall W_i \in S$ do
14:         Find Min$(c(W_i))$
15:     end for
16:     TARGET = $W_i$ with minimum $c(W_i)$
17:     if TARGET = NULL then
18:         Sleep 2s
19:         Jump to Line 4
20:     end if
21:     Return TARGET
22: end procedure
```

5 Experiment Setup

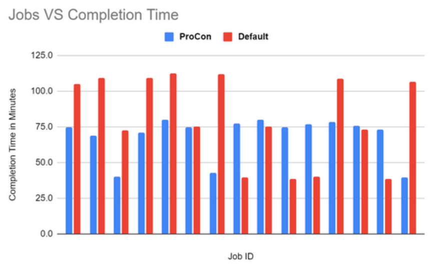
We set up and configured our Kubernetes clusters on Google Cloud Platform[4]. We used a cluster with 4 nodes in total, 16 total CPUs and 128GB of total memory. For our jobs we build an image classification model based on VGG16[5]. The model prints the time consumed for each epoch during training. We built the model as a docker[6] image and pushed it to the Google Cloud Platform. The log analyzer and contention monitor then read the log files from each job and calculate the parameters required for the scheduler. The ProCon scheduler is implemented as a bash script to run on Kubernetes clusters. It directs the incoming jobs to the dedicated worker with the parameters calculated in the previous state.

We evaluate the performance of ProCon with two main criteria: job completion time and makespan of jobs. We submit jobs with two schedules, a fixed schedule and a random schedule. In the fixed schedule, 15 jobs are submitted to the cluster every 60 seconds. In the random schedule, 15 jobs are submitted to the cluster with random intervals.

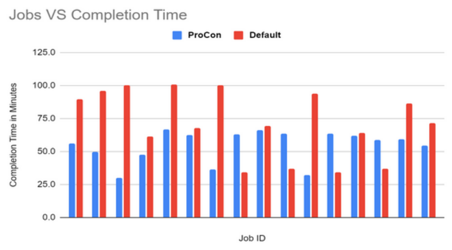
Github Link: `https://github.com/jahnavi0805/ProConScheduler`

6 Evaluation Metrics

Since deep learning applications are computation-intensive , they are more sensitive to CPU than memory spaces and network bandwidth. So, we considered completion time and over all system

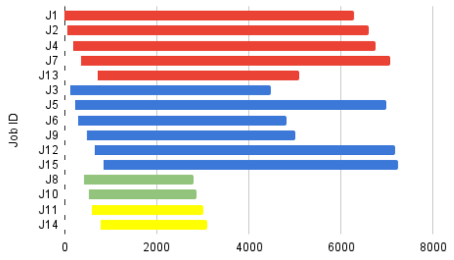


(a) 15 incoming CNN jobs with fixed 60s intervals

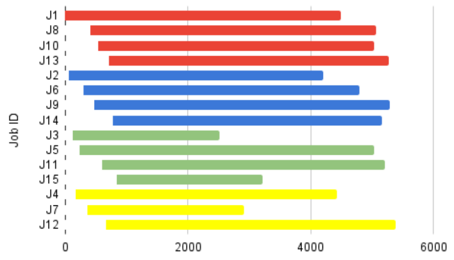


(b) 15 incoming CNN jobs with random schedule(60,250s)

Figure 2: Jobs Vs Completion time for two different schedules



(a) Default with Fixed schedule



(b) ProCon with fixed schedule

Figure 3: Details of the container placement for ProCon and the Default for fixed schedule

performance(makespan and average completion time) as our metrics.To ensure a comprehensive evaluation, we evaluated using fixed schedule and random schedule submission of jobs.

7 Results and Observations

Figure 2.a and 2.b present the job completion time from the experiment. For Fixed schedule jobs, we observed that 9 out of 15 jobs have shorter completion times than default scheduler with an average improvement of 38% and the average completion time of ProCon is 68min compared to default with 81min. We also observed a decrease of 26% in the makespan of jobs from 7236s to 5380s.

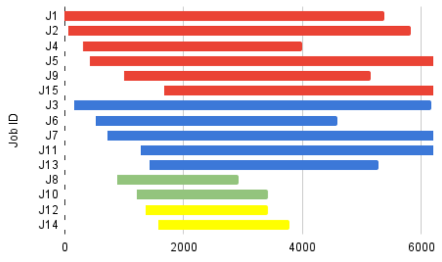
For Random schedule jobs, we observed that 11 out of 15 jobs have shorter completion times than default scheduler with an average improvement of 36% and the average completion time of ProCon is 54min compared to default with 72min. We also observed a decrease of 23% in the makespan of jobs from 7100s to 5438s.

From Figure 3.a and 3.b we find that the distribution with Default scheduler is imbalance since there are 6 jobs on Worker-2 and 2 jobs on Worker-3. from Figure 4.a, 4.b, we observed very similar behavior even when the job submission is random and sufficiently spaced. This is because default scheduler prioritizes the worker based on instant resource usage when the incoming job arrives but ProCon acts like a round-bin algorithm such that each worker is assigned 1 job and then rotate. The standard deviation of the contention rate of worker nodes for default scheduler is greater than ProCon scheduler.

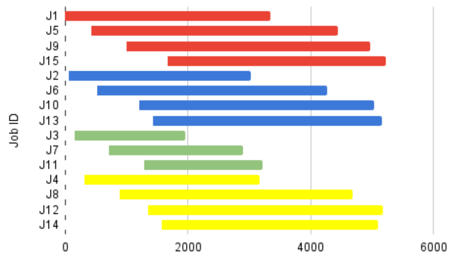
8 Discussion

8.1 Overheads of the experiment

We implement the log analyzer as a daemon process and it is running on all worker nodes. This might include an overhead to our worker nodes. Furthermore the scheduler is implemented as a



(a) Default with random schedule



(b) ProCon with random schedule

Figure 4: Details of the container placement for ProCon and the Default for random schedule

bash script and it is checking for any incoming pods for every 6 seconds. This would add another overhead to the experiment and also waste CPU cycles. As an improvement the scheduler could be written as a plugin and a signaling feature can be added to the scheduler whenever a new pod arrives.

8.2 Mixture of different jobs

Currently only one job type is deployed and submitted to the cluster. For further work we can work on more types of jobs as well as more schedules for jobs. The scheduler is now mainly for the batch jobs with indication of progress in the logs, we could update the scheduler so that it fits a wider range of types of jobs.

References

[1]Y. Fu et al., "Progress-based Container Scheduling for Short-lived Applications in a Kubernetes Cluster," 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 278-287, doi: 10.1109/BigData47090.2019.9006427.

[2] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine

[3] Kubernetes. https://kubernetes.io/.

[4] Google cloud platform. https://cloud.google.com/.

[5] Karen Simonyan and Andrew Zisserman. (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". arXiv:1409.1556 [cs.CV].

[6] Docker. https://docker.com/.