DRAGON: A Dynamic Scheduling and Scaling Controller for Managing Distributed Deep Learning Jobs in Kubernetes Cluster

Chan-Yi Lin, Ting-An Yeh and Jerry Chou
Computer Science Department, National Tsing Hua University, Computer Science Department, Hsinchu Taiwan (R.O.C), Taiwan

Keywords: Deep Learning, Resource Orchestration, Deep Learning, Job Scheduling, Autoscaling.

Abstract: With the fast growing trend in deep learning driven AI services over the past decade, deep learning, especially the resource-intensive and time-consuming training jobs, have become one of the main workload in today's production clusters. However, due to the complex workload characteristics of deep learning, and the dynamic natural of shared resource environment, managing the resource allocation and execution lifecycle of distributed training jobs in cluster can be challenging. This work aims to address these issues by developing and implementing a scheduling and scaling controller to dynamically manage distributed training jobs on a Kubernetes (K8S) cluster, which is a broadly used platform for managing containerized workloads and services. The objectives of our proposed approach is to enhance K8S with three capabilities: (1) Task dependency aware gang scheduling to avoid idle resources. (2) Locality aware task placement to minimize communication overhead. (3) Load aware job scaling to improve cost efficiency. Our approach is evaluated by real testbed and simulator using a set of TensorFlow jobs. Comparing to the default K8S scheduler, our approach successfully improved resource utilization by 20% $\sim$ 30% and reduced job elapsed time by over 65%.

1 INTRODUCTION

To obtain sufficient computing resources for training a large scale deep learning model in a timely fashion, one must rely on distributed training technique on a parallel computing systems, like GPU cluster. A distributed training job is consisted of multiple computing tasks, and each task can utilize a computing device to train a single model together. As shown by a recent study (Goyal et al., 2017), a 50 layers of ResNet model can be trained in 1 hour by using 256 GPUs. Hence, building and managing a resource pool for distributed deep learning training is essential. Furthermore, due to the expensive cost of these high performance computing devices, such as GPU, the resource pool must be shared among users and multiplex between jobs in order to improve overall system utilization and cost efficiency.

The parallel deep learning computing has been extensively studied (Krizhevsky, 2014; Li et al., 2014; Zhang et al., 2016), and it has been implemented in many machine learning computing frameworks, such as TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2017), CNTK (Yu et al., 2014). But these distributed training jobs are commonly managed like the traditional parallel batch jobs or big data applications by the cluster resource managers and job schedulers, such as Kubernetes (Burns et al., 2016), YARN (Vavilapalli et al., 2013), Mesos (Hindman et al., 2011), and SLURM (Jette et al., 2002), where jobs can keep holding on GPUs until their training complete, and each task of a jobs is scheduled independently onto the first available computer slot. As a result, a system could suffer from significant performance degradation and low resource utilization due to the strong dependency and heavy communication among the computing tasks of a distributed training job. Optimizing the performance of distributed training jobs in a shared or cloud resource environment is particularly challenging because of the resource contention between jobs in terms of GPU devices and network bandwidth, etc. Therefore, this work aims to address these issues by designing and implementing a controller in Kubernetes to manage the resource allocation and life-cycle of distributed training jobs.

Kubernetes (K8S) is a open source platform for managing containerized workloads and services. Its main purpose is to orchestrate computing, networking, and storage infrastructure on behalf of user workloads, and its container-centric management environ-

Lin, C., Yeh, T. and Chou, J.
DRAGON: A Dynamic Scheduling and Scaling Controller for Managing Distributed Deep Learning Jobs in Kubernetes Cluster.
DOI: 10.5220/0007707605690577
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 569-577
ISBN: 978-989-758-365-0

ment enables portability and agility across infrastructure providers. It has been widely used in industry production systems. Many deep learning service platforms, including Kubeflow (Kubeflow, 2017), RiseML (RiseML, 2017), Microsoft OpenPAI (Microsoft, 2016), IBM FfDL (IBM, 2018), are also built on top of Kubernetes to help users deploy and launch deep learning jobs on K8S with ease. However, as shown by our study, the current deployment features for distributed training jobs on K8S are at the task level not at the job level. As a result, the existing management functionality, like auto-scaling and load-balancing, cannot be applied to distributed training jobs, and the default K8S FIFO scheduler could cause significant network overhead and even deadlock problem.

To address the aforementioned problems, we proposed DRAGON, named after "**D**eep Lea**R**ning with **A**uto-scale and **GA**ng-schedule **O**n Kuber**N**etes". It is designed and implemented as an extended controller component in K8S to provide the ability of managing distributed training at job level. Therefore, we were able to design more sophisticate scheduling and scaling strategies based on job level or even system level information for performance optimization. Specifically, DRAGON enhances K8S with three capabilities: (1) Task dependency aware gang scheduling to avoid idle resources. (2) Locality aware task placement to minimize communication overhead. (3) Load aware job scaling to improve cost efficiency. As shown by our experiments in a small scaled 2 nodes 8 GPUs cluster, and a larger scaled 4 nodes 32 GPUs simulator, compared to the default K8S manager, our approach significantly improved resource utilization and reduced job elapsed time.

The rest of paper is structured as follows. Section 2 briefly introduce TensorFlow and Kubernetes as the case study subjects in our work, and highlights the problems of existing approaches. The design and implement of our proposed solution, DRAGON, is detailed in Section 3 followed by the experimental evaluations in Section 4. Finally Section 5 gives related work, and Section 6 concludes the paper.

2 BACKGROUND

This work investigates the problems of distributed training on shared and cloud resource environment by studying and optimizing the performance of TensorFlow on Kubernetes. Hence, in this section, we briefly introduce these two software tools and highlight the potential problems and limitations of existing approaches.

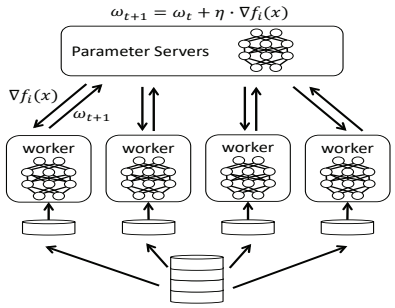

Figure 1: Data parallelism using parameter servers to exchange gradients.

2.1 TensorFlow: Distributed Training

Variant distributed computing techniques have been proposed and implemented for deep learning. In this work, we consider the parameter server strategy implemented by TensorFlow (Abadi et al., 2016) as a study case to investigate the performance of distributed training. We chose TensorFlow because it is one of most popular computing frameworks for deep learning. TensorFlow provides a set of high-level API for users to build their neural network models. The computation on the model is then translated into a dataflow job executed and managed by TensorFlow on any computing devices, including GPUs. Hence it eases the efforts of users to develop and compute deep learning models.

The parameter server implementation of TensorFlow is shown in Figure 1. A distributed training job is consisted of a set of tasks in two different roles. One of the role is parameter server which stores the global parameters of the model and send the current values to each worker for parallel training. The other one role is worker which use the values from parameter server to compute the gradients in each iteration. At the end of each training iteration, workers send the gradients to parameter servers for updating the global parameters before the start of next iteration. The parameter update can be done asynchronously among workers, so the workers can run independently to each other throughout the whole training process and obtain better scalability in large scale training.

The number of parameter servers and workers can be controlled by users, and this setting plays a crucial role to the performance of distributed training jobs. As shown by our experimental results in Figure 2, increasing the number of workers can accelerate the training speed for all three image classification models (InceptionV3, RestNet-50, and AlexNet) we tested. But the speedup is not linear, and the scalability of each model can be different from each other. This is because the communication overhead between

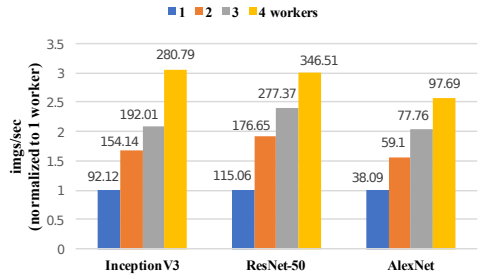

Figure 2: Scalability with images/sec throughput. Higher values means higher performance.

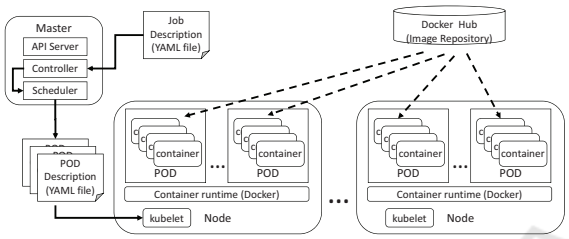

Figure 3: System architecture and components of K8S.

workers and parameter servers for exchanging the gradients and weights of all the parameters after every training iterations. Furthermore, all the processes of workers and parameter servers must be launched before the training can be started by TensorFlow. As a result, distributed training jobs tend have strong task dependency and heavy communication traffic among their tasks. So they should be scheduled as a unit and avoided been placed across compute nodes by the resource managers.

2.2 Kubernetes: Cluster Manager

Kubernetes (K8S) is a open source platform for managing containerized workloads and services on a shared resource pool. It has been broadly used to deploy software across infrastructure providers. The system architecture of K8S is shown in Figure 3. It consists of master nodes and worker nodes. The master nodes host the management components of K8S, and the worker nodes provide the computing resources requested by user's containers. The smallest deployable units managed by K8S is called **POD**, which is a set of tightly coupled containers with shared network namespace and storage among them. A POD can be replicated, recovered, and scheduled by K8S according to its desired state as described by users.

The API server provides the interface for users to interact with K8S. One of the main functions is to let users submit a job description file which contains all the necessary information for deploying and managing the resources of a job throughout its life-cycle, such as container images, resources requirements, storage volume types and so on. The job description file will be handled by a deployment controller, which in turn generates the a description file for each of requested PODs of the job. Then the scheduler is responsible for selecting the worker node for each POD. The default K8S scheduler implements a FIFO queue, and binds PODs to workers in a round-robin manner with the aim of balancing the load among workers. Once the POD requests is received by Kubelet, a node agent on workers, it launches its assigned PODs when there is sufficient residual capacity, and monitors their status and resource usage during execution. Kubernetes is highly reconfigurable and extensible by allowing cluster administrators to define new resource object managed a customized resource controllers. As detailed in Section 3.2, we defined our own resource type for TensorFlow jobs, and implemented our customized controller to optimize job management.

2.3 Challenges

Both TensorFlow and Kubernetes are powerful tools that are broadly used in their respective worlds. But together, there are a few critical problems that could hamper the performance of distributed training in cluster.

First, users must submit a deployment request for each individual workers of a training job. Kubeflow (Kubeflow, 2017) is a recent work that develop a controller to simply the deployment process by generating all the deployment files for users automatically. But it doesn't provide any resource allocation optimization or job life-cycle management ability comparing to our work.

Second, the default K8S scheduler is at POD level, and it tends to spread PODs evenly across all the nodes. But as described in Section 2.1, distributed training jobs has strong dependency among its tasks. So the jobs may suffer from significant communication overhead. Even worse, resources might be wasted when a training task cannot start computing before all its peer tasks are launched. Therefore, both job performance and resource utilization can be suffered when the scheduler is not aware the task dependency of a job.

Last but not least, the existing solution provided by K8S is called Horizontal Pod Autoscaler, which controls the number of PODs for a deployment object according to the CPU utilization of a POD in response to the sudden surge of workload demands. But a single distributed training job can be consisted of multiple PODs, and deep learning computations are of-

ten communication not cmoputation intensive. Therefore, new control policy must be designed for this alternative objective.

3 DRAGON

3.1 Overview

To address the challenges discussed in Section 2.3, we designed and implemented *DRAGON*, which is an extended controller for managing distributed training jobs on K8S. The key design principal of DRAGON is to enhance job scheduling and scaling decisions by utilizing job and system level information. Specifically, DRAGON is based on the following three management strategies for minimizing job execution time and maximizing resource utilization. Detailed implementations and algorithms are given in Section 3.2 and Section 3.3, respectively.

Task Dependency Aware Gang Scheduling: DRAGON is a job level scheduler. It schedules all the tasks of a job as a single unit like a gang scheduler. A job is only launched when the system has enough residual capacity to run all its tasks simultaneously. Thus, resources will not be occupied by idle tasks, and deadlock among tasks can be avoided.

Locality Aware Task Placement: To minimize the communication overhead of a job, DRAGON tends to place all the tasks of a job on as fewer number of compute nodes as possible. A more sophisticate placement algorithm with considerations of network topology and performance interference could also be implemented in our controller. But we use a simple greedy algorithm in this work to demonstrate the capability of our controller. Besides, those sophisticate placement algorithms (Amaral et al., 2017) often required prior knowledge of the job execution time or pre-built performance models which may not be available in practice.

Load Aware Job Scaling: Distributed training are often long jobs with stable workload. But the overall system loading can still varied over time as jobs arrive and leave the system. Hence, DRAGON tends to scale-up jobs when the system loading is low, so that jobs can take advantage of the residual capacity to reduce their execution time. On the other hand, DRAGON tends to scale-down jobs when the system loading is high, so that the resources can be reclaimed from the running jobs to launch the waiting jobs as soon as possible.

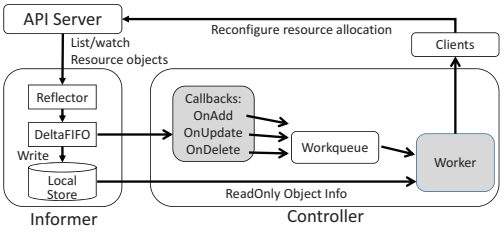

Figure 4: The interaction between a custom controller and K8S. The two gray boxes are the controller-specific components that were re-implemented in our approach.

3.2 Implementation

Our implementation follows the operator pattern defined by K8S. Operator pattern is a way to extend K8S by combining a custom controller along with a custom resource type which specifies the user-defined objects that will be managed by the custom controller. In our implementation, we defined distributed training job as a custom resource, and we implemented a custom controller with the logic to make the scheduling and scaling decisions for these training jobs. With our custom resource, users can describe a distributed training job in a single YAML file, and specify the maximum and minimum number of workers allowed for scaling the job. Hence our approach not only helps users simplify the deployment process like Kubeflow, but also gives users the ability to provide information about how their jobs want to be controlled. For instance, we can easily extend our scheduler to support priority scheduling according to the job priority given by users.

The interaction between a custom controller and K8S is illustrated in Figure 4. Controller relies on another component called informer to register and monitor the events that trigger the controller. In our implementation, we registered all three default setting events (OnAdd, OnUpdata, OnDelete) for our custom resource. OnAdd and OnDelete event indicates a distributed training job arrives or leaves the system. OnUpdate means users manually changes the spec of a job, such as number of workers, job name, etc.

When a registered event occurs on one of the custom resource object, informer adds this object into the workqueue of the custom controller. Then the controller creates a worker process to handle the event on the object according to the control logic implemented by the controller. In our work, we implemented the algorithm described in next subsection. Finally, the controller attempts to change the system state through the command of API server. In our implementation, we use the controller to do job scheduling and scaling by generating new deployment files for the computing tasks that need to be added or changed by our control

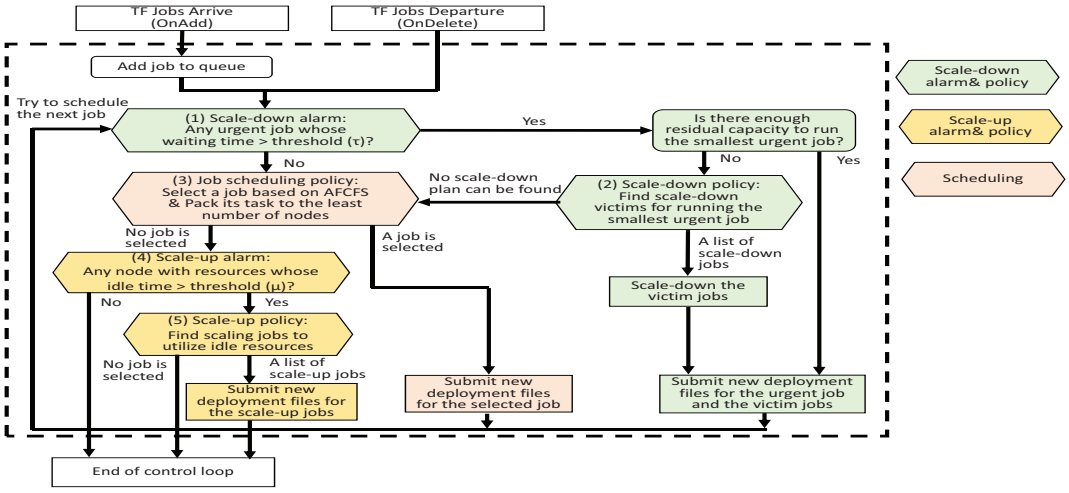

Figure 5: The control flow and policy of DRAGON.

decision.

To enable job level gang scheduling, our controller maintains an internal job queue, and keeps track of current system resource usage. When a job is allowed to be launched, the controller generates deployment requests for each of its computing tasks at once; otherwise, the job simply waits in the queue. Noted, our controller must co-exist with the default K8S scheduler, thus it uses the node selector label in the deployment description file to force K8S scheduler to place computing tasks (i.e., PODs) onto the assigned node locations.

3.3 Algorithm

Here, we introduce our proposed control algorithm that achieves the three design goals mentioned in Section 3.1 under the K8S architecture described in Section 3.2. The overall control flow of our algorithm is shown in Figure 5. The algorithm is consisted of three procedures as follows, and it is called by the controller whenever distributed training jobs arrive or depart in the system.

First, a scale-down procedure that aims to launch urgent jobs in the waiting queue immediately by releasing resources from current running jobs. We define a job is urgent when its queuing wait time exceeds a pre-defined threshold value. We tend to schedule the smallest job with the least amount of resource requirements from the urgent first. If we cannot release enough resources for the smallest job, then the larger urgent jobs certainly cannot be launched as well. Current implementation selects the scale-down victim jobs as the ones with the most surplus resources. Both the scale-down alarm and policy can be re-designed under our control flow for extension.

After the scale-down procedure for urgent jobs, the controller then starts scheduling the rest of waiting jobs with the residual capacity. We implemented an adapted first come first served (AFCFS) scheduling algorithm, which simply selects the first arrival job in the queue whose required resource capacity is less than then residual capacity. To preserve job locality and minimize communication overhead, we pack the tasks of a job onto the least number of compute nodes. This locality aware scheduling algorithm can also be extended with the information provided by the systems or users, such as the network topology, performance interference model, job priority, etc.

Finally, when no job in the queue can be scheduled, the controller tries to further utilize the leftover resource capacity through the scale-up procedure for maximizing resource utilization and minimizing job execution time. However, we also want to prevent jobs suffer from network overhead and reserve some resources for the future arrival jobs. Therefore, the scale-up alarm is only triggered when a resource (i.e., GPU) is idle over a time threshold, and the idle resource can only be allocated to a job that already ran on the same node to maintain job locality.

4 EVALUATIONS

4.1 Experimental Environment

We evaluated our implementation by running real TensorFlow workload on a small scale GPU cluster with 2 GPU nodes connected by 1Gb Ethernet cables. Each node has 4 Nivida P100 GPUs, 16 Intel 2.1GHz CPU cores, and 64GB memory. As sum-

Table 1: Configuration of the jobs. A=AlexNet, I=InceptionV3, R=ResNet50.

Config.	Job1	Job2	Job3	Job4	Job5	Job6	Job7	Job8	Job9	Job10	Job11	Job12
Models	R	I	I	A	R	A	I	R	I	A	I	A
Num. Workers	4	4	2	2	3	2	2	1	2	2	1	1
Min. Workers	2	2	2	2	2	2	2	1	2	2	1	1
Max. Workers	4											
Steps	2800	3600	1000	450	2100	1000	2000	2400	2500	800	1000	1200
Arrival Time	0	0.56s	2.53s	6.34s	7.58s	13.11s	19.01s	27.54s	34.56s	42.58s	47.58s	51.19s

marized in Table 1, the test workload is consisted of 12 distributed TensorFlow jobs that train 3 common-seen image classification models (AlexNet, InceptionV3, ResNet-50) with variant settings in scaling range, training steps, and arrival time. To observe the behavior of our controller under various system loading, the first 5 jobs was generated with a much shorter inter-arrival time than the rest of jobs. The results are presented in Section 4.2.

To further validate the robustness of our approach, we implemented a simulator to evaluate our control algorithm in a larger scale testbed with 4 compute nodes, each has 8 GPUs. The test workload contains 100 randomly generated jobs with Poisson inter-arival times. The mean inter-arrival time is adjust from 62 seconds to 400 seconds to simulate the workload of various loading. The scalability of the jobs is modeled based on our execution time observations from the 3 real NN models used in the real experiments. Each job has its own setting for the maximum, minimum and initial number of workers, and the values were randomly selected between 2 to 8 GPUs. The simulation results are presented in Section 4.3.

Throughout all the experiments, we compare our approach, *DRAGON*, with two other existing scheduling approaches. One is to deploy a distributed training jobs using the Kubeflow toolkit and let the tasks scheduled by the default K8S scheduler. We use the name *Kubeflow* to refer this approach in the rest of this section. The other approach is named *locality-aware scheduler*, which is a simplified implementation of DRAGON without the feature of auto-scaling. So this approach also acts the same as a greedy packing scheduling algorithm that only tends to launch all the tasks of a job on a single node.

4.2 Real Testbed Results

The job running time comparison can be seen in Figure 6. The time of each job is normalized to its results from DRAGON, and noted the y-axis is in log_2 scale. In general, DRAGON and locality-aware scheduler have similar job running time for the first 5 jobs when the system loading is relatively high. But the running time of job6 $\sim$ job12 were reduced by DRAGON be-

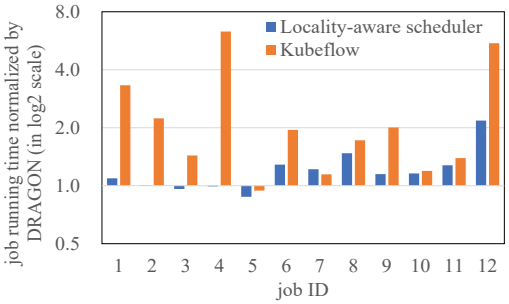

Figure 6: The running time of each job normalized to the results of DRAGON.

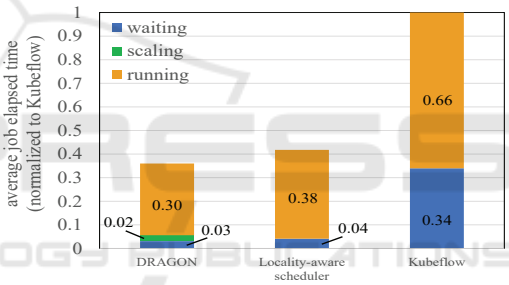

Figure 7: The comparison of the average job elapsed time. The elapsed time is consist of the time for running, waiting and scaling.

cause of the effect of scale-up strategy under low system loading. The job running time of Kubeflow is higher than DRAGON across the board by a factor of 1.2 to 6.3 due to communication overhead.

Figure 7 compares the average job elapsed time which includes the time for running, waiting and scaling. The elapsed time of Kubeflow is significantly reduced by Kubeflow and DRAGON by 60% and 65%, respectively. As shown in the plot, this time reduction comes from both running time and waiting time. The running time is reduced because the network communication time is minimized by the job locality awareness of both locality-aware scheduler and DRAGON. On the other hand, the waiting time is reduced because of the reduction in job running time allows resources to be released sooner, so waiting jobs can be launched earlier as well. In our experiment, more than half of jobs were launched immediately by locality-aware scheduler and DRAGON.

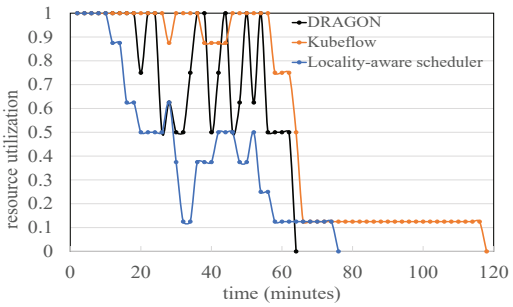

Figure 8: Resource utilization over time.

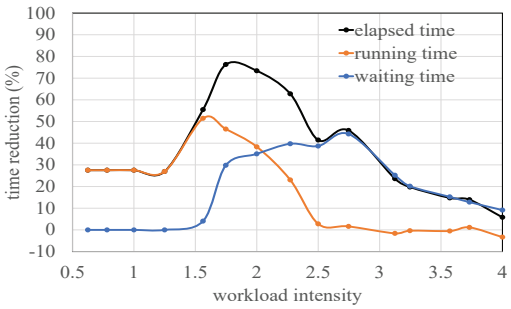

Figure 9: The reduced elapsed time of locality-aware scheduler by DRAGON.

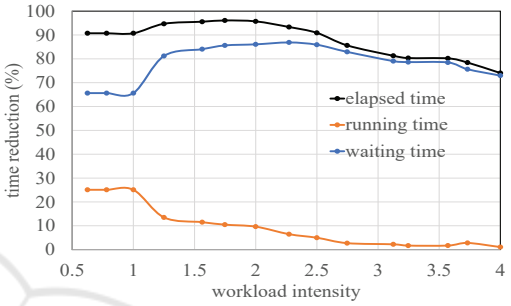

Figure 10: The reduced elapsed time of Kubeflow by DRAGON.

Because of the ability of job scaling, Figure 7 shows that DRAGON has better results than locality-aware scheduler in terms of running time, waiting time and elapsed time. However, scaling job at runtime does cause some overhead to DRAGON because we had to checkpoint and restart the whole job in order to adjust its number of workers. As observed from our real experiment, restarting a job took about 40 seconds to 1 minute. This overhead is still acceptable when the average job running time is long, and the workload variation is not high enough to cause frequent job scaling actions. But, we do think the scaling overhead shoulf be considered and minimized when dynamic resource management approaches are applied, so it will be an important direction of our future work.

Finally, Figure 8 shows the resource utilization over time. Locality-aware scheduler has the lowest resource utilization because it cannot utilize residual capacity unless new job arrives. Kubeflow almost fully utilizes the resources for the first hour because jobs ran much longer and consumes more resources due to communication overhead. But that also causes long job waiting time as shown in Figure 7. Then the utilization of Kubeflow drops extremely low for the next hour because only one job left to run on a single GPU. Hence, the utilization of Kubeflow also can be highly influenced by the system loading. In contrary, DRAGON not only has the shortest job completion time, it is also able to consistently maintain a high resource utilization regardless the variation of system loading. In average, the utilization of DRAGON is 79%, while the utilization of locality-aware scheduler and Kubeflow are 45% and 57%, respectively. So DRAGON achieved higher job execution performance and better resource utilization at the same time.

4.3 Large scale Simulation Results

We use simulator to validate the robustness of our control algorithm under various system workload intensity. Figure 9 and Figure 10 show the amount of time that can be reduced by DRAGON over locality-aware scheduler and Kubeflow, respectively. Clearly, DRAGON was able to reduce the average job elapsed time across a wide range of workload intensity. To analyze the results more deeply, we can also observe that DRAGON has more opportunity to reduce job waiting time under higher system loading by scaling down jobs. However, if the system loading is too high, all three scheduling algorithms suffer long waiting time, so the reduction ratio becomes more limited. Also, the job running time could be increased by DRAGON when jobs are scaled down like a few occasions shown in Figure 9 when the workload intensity is higher than 3. But even under high system load, scale-up strategy could still be triggered to utilize some fragmented resources that cannot be used to launch new jobs.

On the other hand, when system is less loaded, DRAGON scales up jobs to utilize the residual capacity for reducing job running time. But if the loading is low, the jobs will be bounded by the maximum worker setting and even DRAGON cannot utilize all the resources. Therefore, the running time reduction also become less in Figure 9 when the workload intensity is lower than 1.5. By combining the time reduction gains from both running time and waiting time, DRAGON has the ability to reduce the overall elapsed time under any system loading circumstances.

5 RELATED WORK

Over the past decade, great effort has been made from research community to address the scalability and performance problem of distributed deep learning algorithms. Both data parallelism and model parallelism have been extensively studied (Krizhevsky, 2014). For model parallelism, many model partition and device placement strategies have been proposed (Mirhoseini et al., 2017; Mayer et al., 2017). For data parallelism, variant stochastic gradient descent (SGD) algorithms have been developed to reduce communication and synchronization overhead, such as asynchronous SGD (Dean et al., 2012; Zhang et al., 2016), low-precision SGD (De Sa et al., 2017) and lock-free SGD (Niu et al., 2011). Decentralized Parameter server (Li et al., 2014), ring allreduce (Sergeev and Balso, 2018) are the two communication method that have been broadly used for exchanging parameters. Finally, many training skills have been discussed for large scale training, including learning rate, mini-batch size, and the ratio between parameters and workers, etc. For instance, greater mini-batch size is needed to speedup large scale training, but unusually large minibatch size can affect learning accuracy. So (Goyal et al., 2017) proposed a warmup scheme by using less aggressive learning rates at the start of training, and used the LARS (You et al., 2017) algorithm to assign a separate learning rate for each layer instead of each weight. The growing development of these techniques show the need of distributed training, and the importance of managing these distributed training jobs.

Not until recently, people start pay attention to the problem of managing distributed training jobs on parallel systems and shared resource environment. (Amaral et al., 2017) discusses the importance of considering the underline network topology for optimizing communication overhead. It proposed a topology-aware task mapping scheduling algorithm by considering three performance cost: tasks communication, external resource interference, and resource fragmentation. Gandiva (Xiao et al., 2018) optimizes the resource usage for deep learning training by two techniques. One is to dynamically prioritize or kill a subset of jobs according to the early feedback on training accuracy. The other is to time-slice GPUs efficiently across multiple jobs by predicting training performance. Similar to Gandiva, Optimus (Peng et al., 2018) also attempts to minimize job training time based on online resource-performance models. Proteus (Harlap et al., 2017) proposed to the strategy of placing parameter servers and workers on cloud spot instance for cost saving. OASiS (Bao et al., 2018) is a online job scheduling algorithm that computes the best job execution schedule upon the arrival of each job, based on projected resource availability in the future course and potential job utility to achieve. Last but ot least, (Jeon et al., 2018) studies the effect of gang scheduling, multi-tenant GPU sharing, and failures during the execution of training workload on GPU cluster. Same as our work, these studies are motivated by the opportunities and challenges of managing deep learning training dynamically in a shared resource environment. But different from us, these studies aim to propose more sophisticate scheduling and scaling algorithms based on performance prediction models and domain-specific knowledge of deep learning, while we focus more on providing a control mechanism and software architecture to enable dynamic resource management on Kubernetes. Therefore, these studies are complementary to our work, and many of these algorithms can be implemented into our control system.

6 CONCLUSIONS

The demand of distributed deep learning training is growing rapidly in recent years. But deep learning training is a resource-intensive and time-consuming workload that needs to be carefully and dynamically managed in a shared resource environment. In this work, we investigate these performance issues by running distributed TensorFlow training jobs on a K8S cluster, and address the problems by implementing DRAGON, a K8S controller for dynamic scheduling and scaling jobs according to system loading. DRAGON enhances K8S with three capabilities: (1) Task dependency aware gang scheduling to avoid idle resources. (2) Locality aware task placement to minimize communication overhead. (3) Load aware job scaling to improve cost efficiency. The evaluation of our approach was conducted on both real testbed and simulator. Comparing to the default K8S scheduler, DRAGON significantly improved resource utilization by 20% $\sim$ 30% and reduced job elapsed time by over 65% to deliver higher system performance and lower computation cost for distributed training. The design of our controller also allows us to explore the implementations of more sophisticate dynamic resource management strategies in the future. The complete source code of our implementation can be download at (Chan-Yi Lin, 2019).

REFERENCES

Abadi, M. et al. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on OSDI*, pages 265–283.

Amaral, M., Polo, J., Carrera, D., Seelam, S. R., and Steinder, M. (2017). Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proceedings of the SuperComputing (SC)*, pages 17:1–17:12.

Bao, Y., Peng, Y., Wu, C., and Li, Z. (2018). Online job scheduling in distributed machine learning clusters. *CoRR*, abs/1801.00936.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *ACM Queue*, 14:70–93.

Chan-Yi Lin (2019). DRAGON: Deep Learning with Auto-scale and Gang-schedule On Kubernetes. https://github.com/ChanYiLin/tf-operator-Dragon/.

De Sa, C., Feldman, M., Ré, C., and Olukotun, K. (2017). Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual ISCA*, pages 561–574.

Dean, J. et al. (2012). Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, pages 1223–1231.

Goyal, P., Dollar, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. In *arXiv preprint arXiv:1706.02677*.

Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. (2017). Proteus: agile ML elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth EuroSys*, pages 589–604.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. (2011). Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on NSDI*, pages 295–308.

IBM (2018). Fabric for deep learning (ffdl). https://github.com/IBM/FfDL.

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. (2018). Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. *Microsoft Research Technical Report*.

Jette, M. A., Yoo, A. B., and Grondona, M. (2002). Slurm: Simple linux utility for resource management. In *In Proceedings of Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer-Verlag.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997.

Kubeflow (2017). The machine learning toolkit for kubernetes. https://www.kubeflow.org/.

Li, M. et al. (2014). Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on OSDI*, pages 583–598.

Mayer, R., Mayer, C., and Laich, L. (2017). The tensorflow partitioning and scheduling problem: It's the critical path! In *Proceedings of Workshop on Distributed Infrastructures for Deep Learning*, pages 1–6.

Microsoft (2016). Open platform for ai(openpai). https://github.com/Microsoft/pai.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. (2017). Device placement optimization with reinforcement learning. *CoRR*, abs/1706.04972.

Niu, F., Recht, B., Re, C., and Wright, S. J. (2011). Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, pages 693–701.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. (2018). Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 3:1–3:14.

RiseML (2017). Machine learning platform for kubernetes. https://riseml.com/.

Sergeev, A. and Balso, M. D. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799.

Vavilapalli et al. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 5:1–5:16.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L. (2018). Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX OSDI*, pages 595–610.

You, Y., Gitman, I., and Ginsburg, B. (2017). Scaling SGD batch size to 32k for imagenet training. *CoRR*, abs/1708.03888.

Yu, D. et al. (2014). An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report*.

Zhang, W., Gupta, S., Lian, X., and Liu, J. (2016). Staleness-aware async-SGD for Distributed Deep Learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 2350–2356.