Master Thesis, 30 credits

Master Of Science Programme in Computing Science

2021

Abstract

Systems for running distributed deep learning training on the cloud have recently been developed. An important component of a distributed deep learning job handler is its resource allocation scheduler. This scheduler allocates computing resources to parts of a distributed training architecture. In this thesis, a serverless distributed deep learning job handler using Kubernetes was built to compare the job completion time when two different Kubernetes schedulers are used. The default Kubernetes scheduler and a gang-like custom scheduler. These schedulers were compared by performing experiments with different configurations of deep learning models, resource count selection and number of concurrent jobs. No significant difference in job completion time between the schedulers could be found. However, two benefits were found in the gang scheduler compared to the default scheduler. First, prevention of resource deadlocks where one or multiple jobs are locking resources but are unable to start. Second, reduced risk of epoch straggling, where jobs are allocated too few workers to be able to complete epochs in a reasonable time. Thus preventing other jobs from using the resources locked by the straggler job.

Contents

1 Introduction

Deep learning algorithms are getting attention in both research and industry due to successful applications in various domains such as computer vision [6], natural language processing [20], and speech recognition [21]. The rapid growth of data both in volume and variety and increasing model sizes have sparked broad interests to develop distributed machine learning systems to allow for scalable, fault-tolerant and resource efficient training. In this context, distributed machine learning frameworks have been designed to allow for implementation of parallel deep learning training, e.g., MXNet [4] and TensorFlow [1]. Two classes of frameworks have been developed. Multi-tenant, which allows for multiple concurrent deep learning jobs, or single-tenant, which only allows a single job at a time. One of the factors affecting the training time when using such distributed machine learning frameworks, is the framework's resource allocation scheduler [7]. This paper investigated how the training time of distributed deep learning jobs are affected by the type of scheduler used in a multi-tenant distributed deep learning job handler framework using a serverless architecture with Kubernetes[1].

Distributed machine learning training have become popular due to its ability to scale and be more fault tolerant than centralized training [17]. Machine learning training can sometimes use more data than is possible to fit on a single computer. Examples include transaction processing in large enterprises [2], or astronomical data that is impossible to move and centralize [13]. If this is the case, distributed training is the only option if all data is required for training. Additionally, distributed training increases parallelization and I/O bandwidth.

The serverless architecture has recently become of interest for performing distributed deep learning training [19]. The reason for using the serverless architecture for performing distributed deep learning training is that the number of resources used in a machine learning job should be dynamically adjusted. The serverless architecture allows for easy scaling by modifying the number of serverless functions launched. Because machine learning training is a trial and error process, a lot of time is spent modifying parameters and under-utilizing the resources of a traditional cluster. In a serverless architecture, a user only pays for active execution time, and not the idle time between executions.

Using a gang-like scheduler instead of a general-purpose scheduler comes with two benefits. The first of which is that the gang-like scheduler prioritizes jobs in a first-come first-serve manner (Called FIFO in this thesis.). This is important in an online training environment as initially started jobs are expected to complete before later started jobs, if all other things are equal. The second benefit is that the architecture used for distributed deep learning training can cause resource deadlocks if the wrong resources are allocated (Further discussed in Section 3.2.). The gang-like scheduler prevents this resource deadlocking and prioritizes jobs in a FIFO order.

The key to efficient resource scheduling and distributed training is to maximize efficient resource utilization of expensive resources such as GPUs or RDMA networks [12]. Existing resource schedulers are built for general purpose jobs and usually performs static resource allocation [16][18]. Because of this, they are considered sub-optimal for deep learning jobs [12]. Furthermore, properties of deep learning jobs such as high CPU and bandwidth resource requirements are not taken into account by existing schedulers [2]. In this thesis, the default Kubernetes scheduler, that allocates resources evenly, was compared with a gang-like scheduler that allocates resources in a first-come first-serve (FIFO) manner. These comparisons were made to answer which of the two schedulers are most time efficient in a distributed

[1] https://kubernetes.io

[2] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

deep learning training environment.

To be able to compare schedulers, a distributed deep learning training framework, henceforth called Coach, was built together with a gang-like scheduler, called the gang scheduler. Coach is started with one or multiple job metadata files. Coach tries to reach a given loss value with a given model over a number of training epochs. Coach uses Kubernetes as a container manager, and iteratively launches a number of serverless functions that can distributively train the model. Multiple experiments were performed to be able to compare the gang scheduler with the default scheduler. The configurations include the number of concurrent jobs, the deep learning model used and whether Coach was allowed to dynamically scale the number of serverless functions launched or not. To reiterate, the research question is which of the schedulers has the lowest job completion time in a three node cluster with varying number of concurrent jobs.

Our experimental computing cluster performed training using CPUs instead of the more commonly used GPUs[7]. This might affect the results because CPUs are more easily shared than GPUs [12]. Typically, a budget is a part of a deep learning job for DDLTFs, due to time constraints we were not able to run our experiments with a budget constant. This constant, if it was used, would change the behavior of Coach by affecting how many functions are launched. The experiments were performed using two deep learning models, LeNet [9] and ResNet18 [6]. For the ResNet18 model, only half of the training dataset was used because of time constraints. This means that when the loss target was reached, the trained model is not production-ready because its loss on the full data-set would be much lower than the loss value it trained for.

The results did not prove any significant difference between the job completion time when using the different schedulers. However, the gang-like scheduler does prevent resource deadlocks and allocates resources in a FIFO manner. Which makes the custom scheduler more suitable for online deep learning job handling compared to the default Kubernetes scheduler.

2 Earlier Work

The job handler part of this thesis is mostly inspired by two other improvements to distributed deep learning training. The first of which is *Optimus* [12], a dynamic multi-tenant resource scheduler, and the second of which is *Siren* [19], a dynamic single-tenant scheduler using the serverless architecture.

Optimus presents two techniques to dynamically change the number of workers and parameter servers in a parameter server implementation during training. *Optimus* performs online resource allocation by predicting how many training epochs remain until model convergence. The epoch estimation is done by performing a non-negative non-linear least squares algorithm with the input being an epoch number and the loss value for that epoch. *Optimus* also uses the least squares algorithm to estimate the potential gain of allocating resources for an additional worker or parameter. The gain is a function that tries to minimize the remaining training time subject to restraints based on the available resources.

Siren also presents an efficient and dynamic scheduler for distributed deep learning jobs. The biggest difference between *Siren* and *Optimus* being that *Siren* is implemented to work for single tenant systems. I.e. *Siren* is a scheduler for single jobs and not multiple concurrent jobs. *Siren* also uses the serverless architecture because of three reasons:

1. Workers in a distributed training job can be built to be stateless. This makes it possible to use the stateless nature of the serverless architecture to run worker nodes in distributed

training.

2. The number of resources used in a machine learning job should be dynamically adjusted to minimize training time. The serverless architecture allows for easy scaling by modifying the number of serverless functions launched.

3. Because machine learning training is a trial and error process, a lot of time is spent modifying parameters and under-utilizing the resources of a traditional physical or virtual cluster. However, in a serverless architecture, the user only pays for active execution time, and not the idle time between model training.

As for the scheduler, some motivations for its implementation are based on Jeon et al. [7]. They performed analysis of the bottlenecks in multi-tenant distributed deep learning training. They studied three topics: (1) the effect of gang scheduling on GPU utilization, (2) the effect of locality on GPU utilization, and (3) failures during training. They found that locality of training jobs is of high importance because of the high bandwidth requirement of distributed training. I.e. Same-job training nodes should be scheduled in positions that allow for maximal amounts of bandwidth to reduce the risk of data transfer being the bottleneck in training. They also found that it is critical for schedulers to mitigate inter-job interference, they suggest to place jobs on dedicated servers instead of packing *different* small jobs on a single server. A majority of job failures stem from user errors in code, configuration or data. They suggest performing a small error-checking execution of each job to confirm that no such obvious errors exists before allocating expensive resources to jobs.

Previous schedulers have been implemented to improve the performance deep learning tasks in computing clusters. For example, Mao et al. introduced SpeCon, a Kubernetes scheduler specialized for deep learning tasks [11]. SpeCon prioritizes resources to deep learning tasks whose loss-value is decreasing faster over tasks whose loss-value changing slowly. This proved to reduce the completion time of individual jobs in average by 14.8%. However, their scheduler requires additional configuration for each implemented deep learning job by the DL job owner.

In this thesis, a configurationless Kubernetes gang-like scheduler is compared to the default Kubernetes scheduler. The gang-like scheduler is specialized for deep learning jobs, and reduces the risk of resource deadlocks. The scheduler's job completion time is evaluated by using infrastructure that is a combination of *Optimus* and Siren [12][19].

3 Background

3.1 Deep Learning

Deep learning is a sub-field of machine learning concerned with methods based on artificial neural networks. Deep learning uses data to make predictions in the form of classifications or regressions. Deep learning has been used in tasks such as converting text to images [14], or classifying hand-written digits [9]. The main idea behind neural networks, or gradient-based learning is to compute a function $Y^p = F(Z^p, W)$ where Z^p is the p-th input pattern, and W represents the collection of adjustable parameters in the neural network. In a classification task, the output Y^p can be interpreted as the class label of the input Z^p or as a set of probabilities for each class in the output space [9]. A loss function $E^p = \Delta(D^p, F(W, Z^p))$, measures the difference between the ground truth class D^p, and the output class Y^p. The training task is an optimization problem of minimizing a function $E_{train}(W)$ that is the average loss value over a set of labeled examples called the training set $T = (Z^1, D^1), ..., (Z^p, D^p)$ [9].

Most training methods uses a type of gradient descent algorithm [9] to minimize the loss function E_{train}. Gradient descent iteratively adjusts the real-valued weights vector W as follows:

$$W_k = W_{k-1} - \epsilon \frac{\partial E(W)}{\partial W}$$

Where ϵ can be a variable or a constant. One of the most common alternatives to the normal gradient descent algorithm [9] is the stochastic gradient descent algorithm, which updates the weights vector W with a subset of T.

$$W_k = W_{k-1} - \epsilon \frac{\partial E^{pk}(W)}{\partial W}$$

This algorithm fluctuates around an average path towards minimization, but usually converges faster than regular gradient descent algorithms on large training sets with redundant samples. The two most important properties of deep learning training when considering making the training distributed are the property of iterativeness and the property of convergence.

Iterativeness

Model training is most commonly performed in an iterative fashion, due to the complexity of deep neural networks and the large size of training datasets (e.x. 14 million images in the ImageNet dataset[3].) To speed up gradient update steps, training data is usually split up into equal-sized *data chunks*. All data chunks are then further divided into equal-sized *mini-batches*. In each training step, a *mini-batch* is used to update the model weights according to what is calculated as an optimal step by the gradient descent algorithm. For each mini-batch update step, a performance metric is calculated, e.g. a loss (the sum of all errors made for each data point in the mini-batch) or accuracy value (the percentage of correct predictions compared to the labels in a classification model). Each mini-batch training step is called an *epoch*.

Convergence

Convergence means that the decrease or increase in the performance metrics value (loss) between any two consecutive epochs will be very small, ”very small” being defined by the DNN model architect. DNN models are non-convex and are not guaranteed to converge [5]. However, in production ready models, the parameters have often been tuned during an experimental phase which typically allows the model to converge well. Coach is built to work with mature and production-ready models as we leverage the convergence property to make continuous estimations on the number of epochs needed to converge.

3.2 Distributed Deep Learning

Training of deep neural networks is inherently costly and usually requires large amounts of computing resources because of the nature of deep learning. These computing resources can either be stacked on a single computing machine, or split up onto multiple machines. To distributively train deep neural networks, two methods of workload partitioning are most common. Data parallelism, where the training data is split between computing nodes, or model parallelism, where the model is partitioned between computing nodes. For data parallelism,

[3] http://www.image-net.org/

the full model is used by each computing node and for model parallelism, the full training dataset is used for training on each computing node. Both model- and data parallelism are widely used but are considered as sub-optimal in terms of their parallelization utilization [8].

Because each node in distributed deep learning training has to send its gradients over a network for it to be aggregated, deep learning is often bandwidth intensive. To reduce the risk of having the bandwidth of the network being the bottleneck for distributed deep learning, high-bandwidth node-to-node communication is important [19].

Data parallelism

In data parallelism, each node n out of the total number of nodes N uses the same model parameters W for their model $Y^p = F(Z^p, W)$ (see section 3.1), but different disjoint subsets of the training dataset T^n such that $\cup_{n=1}^{N} T^n = T$. Each worker performs a gradient update step with its dataset and can either wait until all other workers have completed their gradient update step, or continue working asynchronously. If a synchronous data parallel method is used, the gradient updates from all workers are collected before a new gradient update can be performed on any worker. The collected gradients are then aggregated. Once the gradients have been aggregated, the resulting gradient is sent to all workers and the next gradient update step is started.

In an asynchronous data parallel implementation, workers can continue performing gradient update steps asynchronously. This works because gradient aggregation is performed on each completed gradient update step by each worker. That is, whenever a worker completes a gradient update step, the gradient is aggregated with all completed gradients of the same update step, the worker then continues with its next update step after having received the update gradient.

Parameter server architecture

The requirements of data parallel deep learning training is that each gradient from each worker per epoch must be stored on a single node for all gradients to be aggregated. In the parameter server architecture, computing nodes are split up into workers and parameter servers. A worker performs gradient updates based on its subset of the dataset and sends the updated gradients to a parameter server. The parameter server receives all gradients from all workers and performs a gradient aggregation for each epoch (in synchronous training). Multiple parameter servers can be used to improve scalability, redundancy and availability by migrating or replicating the aggregated gradients. That is, if a parameter server crashes, another one can replace it without having to restart the job.

The MXNet (used for the experiments of this article) implementation of the parameter server (PS) architecture uses parameter *servers*, *workers* and a distributed training manager called the *scheduler*. The *scheduler* is responsible for server and worker discovery. All PS types must first register with the *scheduler* and are then provided the connection details for the other *servers* and *workers*. Before a PS job is started, the number of *workers*, $w \in \mathbb{N}$ and *servers* $s \in \mathbb{N}$ must be specified. And before training starts in a PS job, each *worker* and *server* must register to the *scheduler*. If the *servers*, *workers* and the *scheduler* requires scheduling, it is crucial that at least one of each type is scheduled to allow the job to start. If any one PS type is not allocated for, the resources allocated to the other PS types will be locked and the job will not be able to start, this resource locking is henceforth called a *parameter server resource deadlock*.

3.3 Serverless Architecture

Since the beginning of software development we have witnessed a transition from larger monolithic applications to smaller applications with a more focused functionality [15]. This evolution of software development is captured by the popularization of various software architectures, such as, service-oriented architectures or micro-service architectures. A more recent symptom of this evolution to smaller applications is visible in platform architectures. Platform as a service (PaaS) is a type of cloud computing service where a cloud provider delivers a platform to clients, allowing them to develop, run and manage applications without the need to maintain software infrastructure. PaaS is typically provided in the form of hosted physical or virtual machines and allows for long-running services such as website hosting. Functions as a service (FaaS), or serverless architecture is much more small grained than PaaS, and hosts applications that are only run when invoked. The benefits of hosting a serverless application compared to a server(full) application (when using a FaaS service such as AWS Lambda[4].) is that the client only pays for when the function is invoked, instead of all the running time. This is also a clear benefit for the platform/functions provider in that resource usage is more efficient. Multiple functions can be hosted on the same node and be invoked in an interleaved fashion such that each functions has access to the resources it requires.

On the current FaaS platforms, limitations are set for the types of functions that are allowed in the maximal allowed run time of the functions (15 minutes on AWS Lambda[5]), or the amount of memory allowed (max of 10 GB on AWS Lambda[6].).

3.4 Containers

A container is a packaged piece of software that contains code for an application and all dependencies necessary to run said application. This allows for quick setup and startup of the application on multiple types of computing environments. Containers can be seen as lighter versions of virtual machines in that they are isolated, but does not use a hypervisor for operating system access. Instead, containers have direct access to the host operating system. It's important to note that containers can, and are usually run inside virtual machines.

3.5 Kubernetes

Kubernetes[7] is a tool for deployment, scaling and management of containerized applications. To manage a container with Kubernetes, a cluster needs to be created. A Kubernetes cluster consists of nodes and pods, where a node can host multiple pods. Each pod is tightly coupled with some shared resources (network interface, storage, GPU, CPU etc.). A pod is scheduled to execute on one physical machine and is a group of one or more containers. Scalability and fault tolerance is a feature of Kubernetes in that the same pod can easily be replicated across multiple physical machines. A node is an abstraction of a physical *or* virtual machine that allows for hosting of pods. Each node is managed by the Kubernetes control plane and contains everything necessary to run one or multiple pods.

[4]https://aws.amazon.com/lambda/ 2021-05-12

[5]https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/

[6]https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/

[7]https://kubernetes.io/

Kube-scheduler

Scheduling in Kubernetes is a task of choosing *placement* of pods to nodes in a Kubernetes cluster. A *placement* is a partial, non-injective assignment of a set of pods to a set of nodes. The scheduling task is an optimization problem, where the scheduler determines a set of *feasible placements* that meet a set of constraints. After having decided the set of *feasible placements*, the scheduler must filter the *feasible* set into a *viable* set of nodes. The *viable* set is a subset of the *feasible* set based on which nodes have the highest scores out of the *feasible* set. The scheduling process is a multi-step process, where one pod is scheduled per iteration.

This Node selection process for a pod can be abstracted to a two-step process:

1. Node filtering
2. Node scoring

The node filtering step finds a set of nodes where it is feasible to schedule the current pod. For example, each candidate node must at least meet the resource requirement of the scheduled pod. Each node that does not meet the resource requirement is therefore filtered. After the filtering step, the list of candidate nodes only contains *suitable* nodes. If the set of *suitable* nodes is empty, the pod is not yet schedulable.

If there are more than one *suitable* node, the scoring step is used to rank the *suitable* nodes according to which node is most *suitable*. If there are more than one node with the same score, the scheduler selects one of these at random.

The Kube-scheduler is built to be easily extendable in the form of creating plugins for any step of the scheduling process. As most types of jobs differ in resource requirements, they might not be optimal with the default Kubernetes scheduler. A custom plugin might improve performance by increasing scheduling speed, node placement or resource usage.

3.6 Nuclio

Nuclio is a high-performance *serverless* framework suitable for data, I/O, and compute heavy workloads. The Nuclio framework allows for simple building and deployment of efficient serverless functions in multiple languages. Nuclio is most easily compared with function-as-a-service (FaaS) providers such as Google Cloud functions or Amazon AWS Lambda. The biggest difference between Nuclio and these FaaS providers is that Nuclio allows for execution of GPU and CPU resources, instead of only allowing for execution on CPU resources.

4 Systems Description

This section presents a technical overview of *Coach*'s job handler, the *gang scheduler* and the *default scheduler*. The job handler is responsible for receiving job requests including a budget, a deep learning model in the form of a python MXNet[8] implementation and an optional target loss value. The scheduler is a modification of the Kubernetes default scheduler, which tries to schedule pods of the same job together.

[8] https://mxnet.apache.org/versions/1.8.0/api/python

4.1 System Overview

A distributed deep learning job requires some number of worker and parameter server functions to be launched. To allow for cost and time efficient worker and server allocation, some estimations about the cost and time of epochs are required to reach model convergence.

To be able to effectively decide the number of servers and workers to launch for each job and epoch, Coach runs four mini-epochs, taking 20-40 seconds to complete for each job. These mini-epochs uses statically set number of workers and servers. The mini-epochs allows Coach to get an idea of how the epoch runtime is affected by the number of workers or servers, as well as what an epoch would cost with various number of servers and workers. Information from all completed epochs (including the mini-epochs) are used by Coach to set what Coach believes to be an optimal server and worker count, based on the available budget and epoch history. This worker and server count selection does not regard to the available computing resources.

After a worker and server count have been selected without regards to the available resources, the worker and server count can either be accepted or reduced based on the available resources. This filter is done iteratively by using the marginal utility function (described in Section 4.2) which tries to maximize efficient resource usage and remove the risk of giving too small a dataset to each worker. When the marginal utility step has been performed, the functions are requested to be deployed. Deployment requests do not guarantee pod deployment, as a request can time out. The Kubernetes scheduler decides which deployment requests get approved, and which requests must wait for additional resources to become available. For an epoch to be able to start, at least one worker, one server and one MXNet scheduler must be successfully deployed. If the minimal requirements are met, but not all deployment requests for an epoch have been approved, a timeout can occur which starts the epoch with the currently available workers and servers. After an epoch has completed, the resulting loss value, runtime and cost are saved, and the process restarts from the worker and server count selection. Epochs are run until the target loss value (or less) has been reached three epochs in a row.

4.2 Marginal Utility Estimation

Any function invocation F uses w and s number of workers and parameter servers ($w, s \in \mathbb{N}$). Each worker can either share resources (CPU, memory, bandwidth etc.) or be allowed exclusive access to the available resources on a physical machine. If each worker is allowed exclusive access, the speedup of the local gradient update step will scale near-linearly to the number of workers selected for epoch invocation F [10]. That is, if w increases, the runtime of F decreases. However, if any two workers $w_i \in w$ and $w_j \in w$ shares computing resources, resource blocking can occur which will reduce speedup, or even produce slowdown of F. To maximize resource usage while reducing the risk of resource blocking, Coach uses marginal utility estimation for w and s selection.

To determine if Coach should invoke an additional function, Coach uses the least squares method to fit the following function:

$$s = -\beta_1 \cdot (w - \beta_2)^2 - \beta_3 \cdot (p - \beta_4)^2 + \beta_5$$

In this function w and s stands for workers and parameter servers, respectively. While $\beta_1, .., \beta_4$ are coefficients that will be approximated with the least squares algorithm. The left-hand variable e stands for the number of epochs completed per second, $epochs/s$. We call this

function $f(w, s)$. The marginal utility of adding a worker to the epoch would then be $f(w, s) - f(w + 1, s)$.

This function also allows Coach to make decisions regarding if the next function to deploy should be a worker or parameter server. This is done by simply checking which of a server or worker would provide the larger marginal utility.

4.3 System Architecture

This section aims at removing any ambiguity of how Coach works. Coach is a standalone program that is started outside the Kubernetes cluster. Coach must be able to request pod deployments, and Coach must be able to communicate with deployed pods using Nuclio. These steps are depicted in Figure 2. The steps depicted in Figure 1 are the main parts of the system and are described in more detail below.

Coach is started with a number of job descriptions. These descriptions include a budget, a target loss and a path to a deep learning model written in python. The first step of Coach (2. in Figure 1.) is to parse all job files into a queue. The queue is then iterated in step four to get data on the cost and run time of an epoch of any job. After iteration, Coach uses the data from step four to calculate an optimal number of functions to invoke for each job, this includes minimally one of each parameter server type as describe in Section 3.2. This optimal number might not be optimal, as the marginal utility function must be used to prune this number. This pruning is done in step six of Figure 1. In between step six and seven, the functions are requested for deployment to the Kubernetes scheduler. Step seven is not started until one of each parameter server have been allocated by the Kubernetes scheduler. Once the functions have been invoked. Coach awaits response from the workers. Step 3-9 occurs asynchronously for each job. That is, for any two jobs, A and B. Job A can be at step x, while job B can be at step y, $x \in \{3..9\}, y \in \{3..9\}$.

As for the included components and their responsibility seen in Figure 2. Coach is responsible for keeping track of job progress, invoking jobs and requesting pod deployments. Coach uses Kubernetes to manage the deployed pods (deleting and creating pods.) Coach also uses Nuclio to invoke pods. Kubernetes uses the Kubernetes scheduler to allocate resources to pods. The pods contain the deep learning model, and a Nuclio wrapping that allows for function invocation. A finished function invocation responds to Coach with a loss value or an error message.

4.4 Default scheduler

Each newly created pod has different resource requirements, which the default Kubernetes scheduler tries to satisfy by allocating what it believes to be the *optimal* node to each pod. The default Kubernetes scheduler takes into account metrics of a pod when scheduling such as individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and so on[9]. The default scheduler considers all pods equal and can not take into account the necessity of gang scheduling for parameter server deep learning training (The necessity as described in Section 3.2.). That is, it does not discriminate against any resource allocation request, unless the resource requests are given specific weights. Which means that if only two resource slots are available, and two parameter server setups are requested for allocation (3x2 number of pods). Any choose three out of six combinations of the 6 different pods are possible for

[9] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

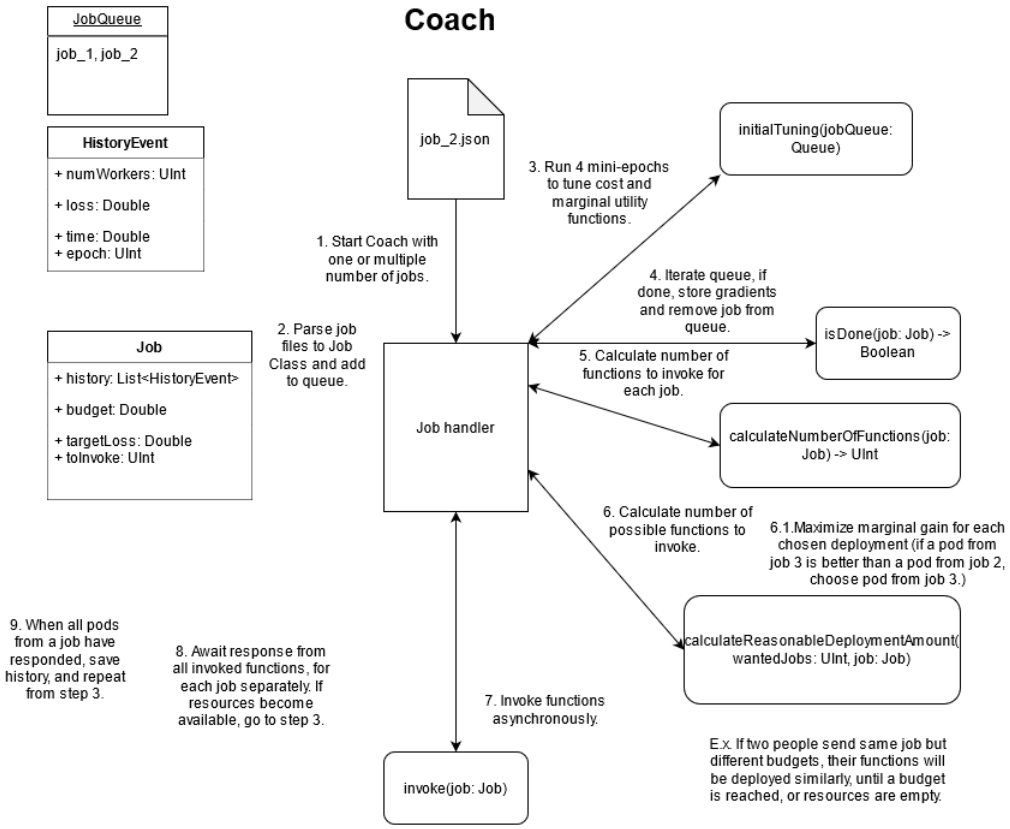

Figure 1: A flow diagram of Coach, a serverless distributed deep learning job handler. Coach is started with a number of job files and performs the depicted steps to reach a given loss value while trying to stay within a given budget.

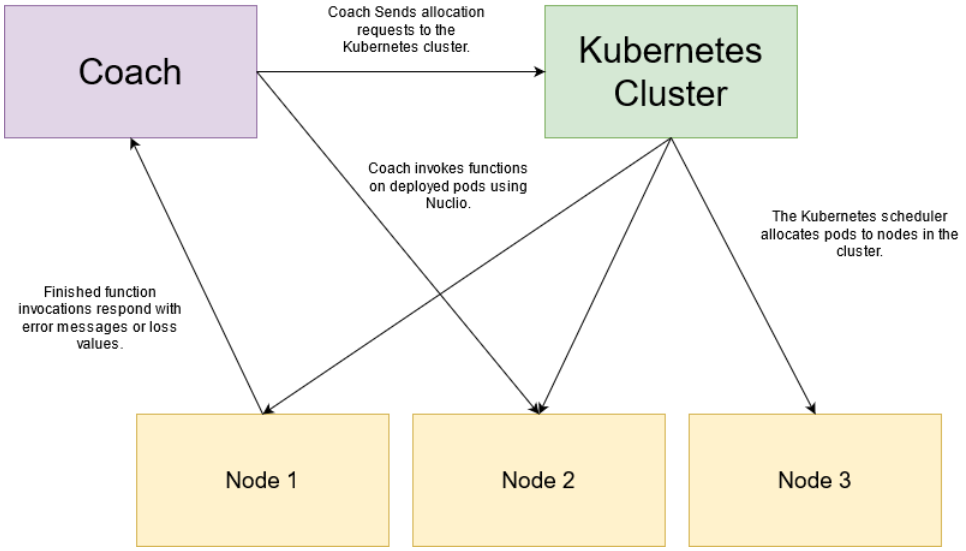

Figure 2: An example overview of the components included when using Coach, a serverless distributed deep learning job handler. The main components include the Kubernetes cluster, the scheduler, Nuclio and Coach. These components allow for distributed training of deep learning models on a serverless architecture.

allocation by the default scheduler. This is assuming all pods are requested for scheduling simultaneously.

4.5 Gang scheduler

The gang scheduler is a fork of the default scheduler with modifications in the pod queue, filtering and the node scoring plugins (see Section 3.5.). The goal of the gang scheduler is to prioritize all pods from job a to be scheduled before any pod from job b if job a scheduled *any* pod before job b. That is, for any two jobs a and b, if job a scheduled *any* pod before job b scheduled any pod, all pods from job a should get resource priority over pods from job b. If this goal is implemented, it should reduce the training time of the oldest job Coach has received, compared to if the default scheduler is used because resources are prioritized to older jobs before newer jobs. Additionally, because the parameter server architecture requires large amounts of bandwidth between worker and server nodes, it is important that parameter servers and worker nodes are scheduled on the same physical machines and not on different machines. This is because the bandwidth between processes on the same physical machine is limited only by the speed of shared memory reading, while computer to computer communication includes the overhead and (much) lower bandwidth of the network layers [3].

While this implementation is not a *true* gang scheduler in that it does not guarantee all-or-nothing scheduling of all pods in a scheduling ”gang”, it is similar enough to a gang scheduler and does provide some gang-like properties. For simplicity’s sake, this scheduler will henceforth be called the gang scheduler.

To allow for these goals to be implemented in a custom Kubernetes scheduler, the Kubernetes scheduler was forked, and three plugins were modified. First, the pod queue plugin, which sorts pods-to-be-scheduled in a scheduling queue. Pods closer to the front of the queue are prioritized resources over pods further from the front. Second, the node filtering plugin, which either accepts or rejects a pod and node pair, and finally the node scoring plugin which is the scoring step describe in Section 3.5.

1. Pod Queue

The queue of pods-to-be-scheduled is sorted according to job creation time. That is, for any two job creation times (defined as the time in seconds when the first pod from the job is requested to be scheduled.), $A \in \mathbb{R}$ and $B \in \mathbb{R}$, where A is before B. Any pod created at time A will be scheduled before any job created at time B. This queue will allow for maximal resource allocation to older jobs before newer jobs. All resources should first be provided to jobs created at time A before being allocated to jobs created at time B. If $A == B$, pods are ordered by their pod name in alphabetical order. Each pod in the to-be-scheduled queue are sorted according to Algorithm 1.

Algorithm 1: The pod queue plugin implementation.

Input: pod1, pod2 to be compared
Result: If pod1 is to be prioritized over pod2 in the pod queue.
jobTimeStamp1 = getTimestampOfJobFrom(pod1);
jobTimeStamp2 = getTimestampOfJobFrom(pod2);
if *pod1.isPriortizedOver(pod2* **then**
  return True
if *pod2.isPrioritizedOver(pod1)* **then**
  return False
if *jobTimeStamp1.equals(jobTimeStamp2)* **then**
  return getName(pod1) ¡ getName(pod2)
return jobTimeStamp1.isBefore(jobTimeStamp2)

2. Node filtering

To prevent resource deadlocks where not all pod types are scheduled for a job and the resources becomes unavailable, the filtering plugin was changed. The filtering plugin decides which nodes are *feasible* and which are not. All the default filters are concurrently active with the gang scheduler filter. The modified filter stores which pod types have been scheduled for each job, and only allows a pod to be scheduled on a node if all job types for that job have been added to the scheduling queue or have already been scheduled. The node filtering modification can be seen in Algorithm 2. The filter plugin modification in combination with the pod queue plugin modification will make sure that all pod types (scheduler, worker, server) required for an epoch to start will be scheduled together because of:

- Deep learning pods for job A will only be scheduled if at least the scheduler, worker and server for job A have been added to the scheduling process.
- If the first pod of job A was added to the scheduling process before any other job B, at least one server, worker and scheduler will be allocated resources and the epoch will eventually run and complete.
- If the first pod of job A was added to the scheduling process after some other job B, job B will be prioritized and allocated resources until it has completed its job, after of which job A will be allocated resources before any job C which was first scheduled after job A.

Because the job handler never deletes all pods, there will always exist one pod of each type,

if they have been allocated before.

Algorithm 2: The node filtering plugin implementation.

Input: *pod* and *node* pair
Result: If *node* is *feasible* with *pod*.
if *not pod.IsDeepLearningPod()* **then**
  return True
addJobTypeToJobTypeMap(*pod*.dlType)
return allJobTypesHaveBeenScheduledFor(*pod*.jobID)

3. Node scoring

A score value is provided to each node for each pod that can be allocated to said node. A pod will be allocated to the node with the highest score for that specific pod. Coach uses a score that prioritizes allocating pods from the same job on the same nodes before allocating pods from different jobs. That is, if there are two nodes, x and y, each providing resources to pod p_1 and p_2 from the set of pods from different jobs j_1 and j_2 respectively ($p_1 \in j_1, p_2 \in j_2$ and $p_1 \in x, p_2 \in y$). If a new pod, $p_3 \in j_1$, is being scheduled, p_3 will be hosted on x rather than y if all other metrics are equal (available node resources etc.). This node scoring is implemented according to Algorithm 3.

Algorithm 3: The node scoring plugin implementation.

Input: pod with node to be scored.
Result: The score $S \in \{0, 100\}$ of node with pod.
`/* Max score for non-deep learning related pods.  */`
if *not pod.IsDeepLearningPod()* **then**
  return 100
if *node.differentJobPodExists(pod)* **then**
  return 0
return 100

This modification is motivated by two facts. The first of which is that process to process (In the same computer.) communication should have higher bandwidth than communication between computer (RAM bandwidth VS. network bandwidth). The second of which is that different job pods will *never* communicate, while same job pods might communicate (servers and workers transfers gradients.). If a server and worker from the same job are scheduled on the same node, they will have higher bandwidth than if they are scheduled on different nodes. If two different workers are scheduled on the same node, they will compete for bandwidth. If a server and worker from different jobs are placed on the same node, they will also compete for bandwidth. This modification reduces the risk of placing different job pods on the same node.

5 Method

To be able to compare the job run times of the Kubernetes schedulers, different experimental setups were used. The different configurations consisted of the following:

- Two different deep learning models, LeNet and ResNet18.
- The scheduler used, gang-like or default.

- Static or dynamic number of resource allocation requests.
- Number of concurrent jobs.

The exact configurations used in the experiments can be seen in Table 1. These configurations will provide insight into how the schedulers perform with varying workloads. The configurations and the computer specifications are described below. All configurations were run once, except one configuration which was run five times per scheduler. The goal of running one of the experiments multiple times was to get statistical support of the results.

For the experimental set up a Kubernetes cluster was used. The Kubernetes cluster consisted of three nodes. All nodes having the same specifications:

- 32x Opteron 62xx-cores
- 56 GB ram
- 4 raid10 HDDs
- Ubuntu 20.04[10]

The nodes were connected with gigabit networks with a theoretical maximum of 1Gbit/s bandwidth, but an actual 110MByte/s bandwidth confirmed by bandwidth testing. All nodes acted as worker nodes (see Section 3.5), and one node also acted as master node and control plane. Each node used Kubernetes version 1.21[11].

The deep learning models were selected because of their time to completion and maturity. The first model was the LeNet [9] model on the MNist hand digit dataset[12], and the second model was the ResNet18 model [6] with a subset of the Cifar10 dataset[13] ($D = x \in C : |D| = 25000$, D is the dataset used, and C is the full Cifar10 dataset.). The target loss for the respective jobs were selected based on time constraints while still allowing for discussion and analysis based on the job completion time. That is, the LeNet job had a target loss of $L_l \in \mathbb{R}, L_l = 0.83$, while the ResNet18 job had a target loss of $L_r \in \mathbb{R}, L_r = 0.5$.

During runtime, Coach collected a number of metrics for each job. The metrics collected included the number of epochs taken ($e \in \mathbb{N}$) to reach $L_x \in \{L_l, L_r\}$ three times in a row. The time taken from job start until job completion $t_t \in \mathbb{R}$ and the time taken for each separate epoch, also $t_e \in \mathbb{R}$. The number of workers and servers deployed for each epoch was also recorded, represented as $w \in 1, 2$ and $s \in 1, 2$. These runtimes were used to compare the schedulers, as the only difference between two similar experiments is the type of scheduler used.

The experiments were run by first making sure all unnecessary pods were deleted from the Kubernetes cluster, and then starting Coach with different configurations described below. Coach was either run with one, two or three jobs, each job being identical to the other jobs because of simplicity. Each job configuration was also run with a static number of workers or servers (See Section 3.2.). All the configurations that were run are shown in Table 1. The configuration using the LeNet model with three concurrent jobs and dynamic w and s selection was run five times for each scheduler for statistical purposes.

[10]https://wiki.ubuntu.com/FocalFossa/ReleaseNotes
[11]https://kubernetes.io/blog/2021/04/08/kubernetes-1-21-release-announcement/
[12]http://yann.lecun.com/exdb/mnist/
[13]https://www.cs.toronto.edu/~kriz/cifar.html

Table 1 The different experiment configurations.

Model	# Jobs	Static/Dynamic	Scheduler
LeNet	1	Dynamic	Co-Scheduler
LeNet	2	Dynamic	Co-Scheduler
LeNet	3	Dynamic	Co-Scheduler
LeNet	1	$w = 2, s = 2$	Co-Scheduler
LeNet	2	$w = 2, s = 2$	Co-Scheduler
LeNet	3	$w = 2, s = 2$	Co-Scheduler
LeNet	1	Dynamic	Default
LeNet	2	Dynamic	Default
LeNet	3	Dynamic	Default
LeNet	1	$w = 2, s = 2$	Default
LeNet	2	$w = 2, s = 2$	Default
LeNet	3	$w = 2, s = 2$	Default
ResNet18	1	Dynamic	Co-Scheduler
ResNet18	2	Dynamic	Co-Scheduler
ResNet18	3	Dynamic	Co-Scheduler
ResNet18	1	$w = 2, s = 2$	Co-Scheduler
ResNet18	2	$w = 2, s = 2$	Co-Scheduler
ResNet18	3	$w = 2, s = 2$	Co-Scheduler
ResNet18	1	Dynamic	Default
ResNet18	2	Dynamic	Default
ResNet18	3	Dynamic	Default
ResNet18	1	$w = 2, s = 2$	Default
ResNet18	2	$w = 2, s = 2$	Default
ResNet18	3	$w = 2, s = 2$	Default

6 Results

The gang-scheduler and default Kubernetes scheduler were compared according to the method described in Section 5. The scheduler software implementation is available from the author[14]. To investigate the run-time differences between the two schedulers for distributed deep learning jobs, 12 computational experiments were performed, as described in Section 5 and shown in Table 2. The experiments are numbered according to the identification numbers in Table 2, column one named Test ID. The experiments were constructed in order to highlight when the gang scheduler might be preferred over the default scheduler, and when it might not be preferred. Each experiment from Table 2 was run once, except experiment #11 which was run five times for each scheduler.

The experiments shown in Figure 3 are further detailed in Table 2, where the plot ID equals the test ID in column one. The runtime is measured from when Coach has received the number of jobs necessary for the experiment (which occurs on Coach startup), and until all jobs have reached their target loss value. When a job is finished, its pods are not killed, and they are not invoked. All jobs of a single experiment uses the same model, dataset, target loss, container image and budget.

Figure 4 uses the same identifiers (1-12) as Figure 3 and is as such also further detailed in

[14]https://github.com/Frans-Lukas/DLScheduler

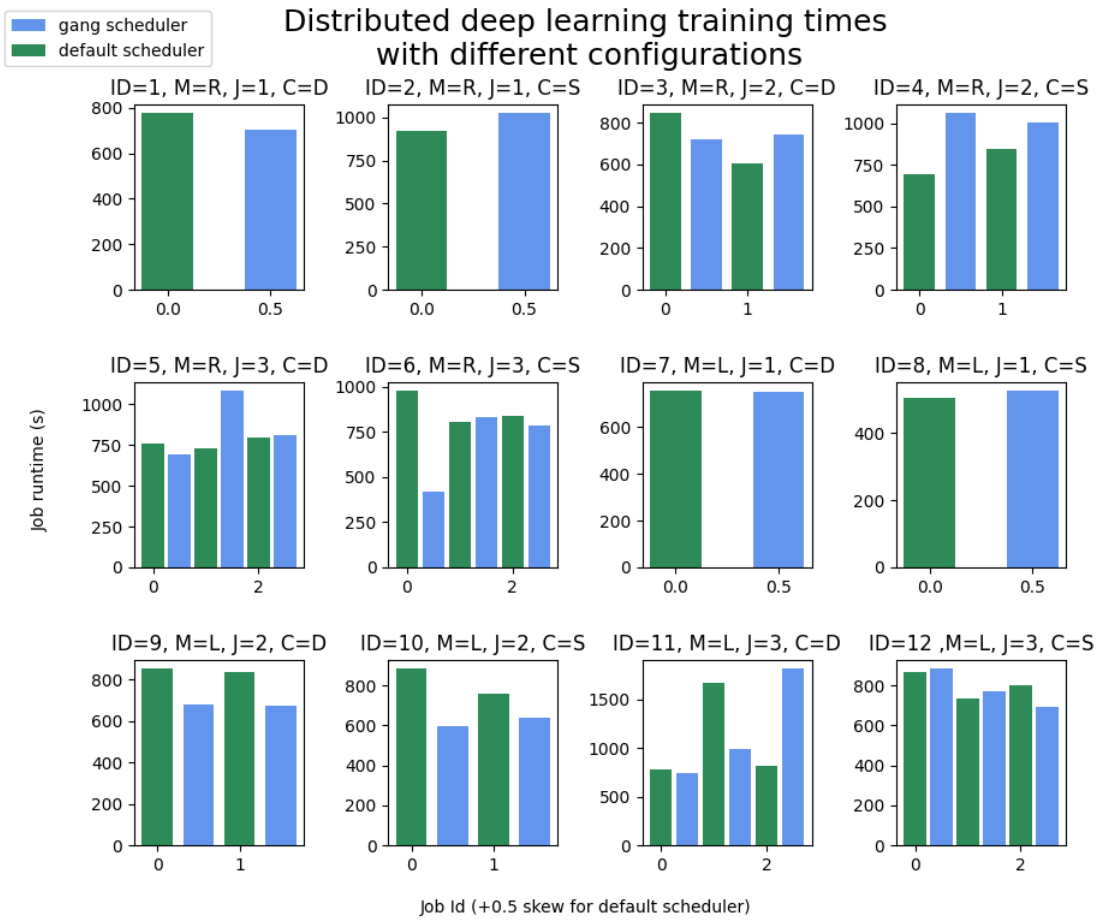

Figure 3: Run-time of the distributed deep learning training jobs with different configurations and schedulers using a multi-tenant job handler. The configurations include different deep neural network models with datasets, dynamic or static worker and server count and multi- or single-tenant workloads (up to three concurrent jobs). All jobs in the same bar-plot uses the same model and dataset, but uses different schedulers. M is the model and dataset used, J is the job count and C is whether the number of workers and servers are dynamically changed or static. If $C = S$, the static count is set to two for workers and servers. $M \in \{ResNet18, LeNet\}$, $J \in \{1, 2, 3\}$, $C \in \{D, S\}$. The job ID (x-axis) exists to be able to tell the jobs apart in the plot. All jobs of the same plot are of the same configuration (model, dataset, target loss and budget).

Table 2 The different experiment configurations and mean runtime results seen in the bar plots in Figure 3. The test ID starts from the top left corner and increments in reading order. Each configuration was run with the Default- and the Gang scheduler.

Test ID	Model	# Jobs	Static/Dynamic	Runtime (s) Default (job mean)	Runtime (s) Gang (job mean)	Default time - Gang time
1	ResNet18	1	Dynamic	777	706	+71
2	ResNet18	1	$w = 2, s = 2$	926	1025	-99
3	ResNet18	2	Dynamic	724	732	-8
4	ResNet18	2	$w = 2, s = 2$	770	1031	-261
5	ResNet18	3	Dynamic	760	859	-99
6	ResNet18	3	$w = 2, s = 2$	872	678	+194
7	LeNet	1	Dynamic	755	752	+3
8	LeNet	1	$w = 2, s = 2$	507	526	-19
9	LeNet	2	Dynamic	845	676	+169
10	LeNet	2	$w = 2, s = 2$	822	617	+205
11	LeNet	3	Dynamic	1089	1184	-95
12	LeNet	3	$w = 2, s = 2$	801	783	+18

Table 3 The statistical measurements in detail as seen in the boxplot in Figure 5. All columns except count and scheduler type are in seconds.

Scheduler type	count	Mean	std	min	25%	50%	75%	max
Default	15	1273.95	446.77	738.80	831.29	1552.73	1691.09	1750.79
Gang	15	1130.03	469.39	737.62	783.20	860.37	1687.86	1845.00

Table 2. Take notice of that the x- and y-axes are of varying scale for both Figure 3 and Figure 4. For example, the y-axis of plot 5 of Figure 4 varies from 20 to 100 seconds, while the y-axis of plot 8 varies from 0 to 30 seconds. Scatter plots for the individual scheduler configurations can be seen in the appendix, Figure 6 and Figure 7.

A total of five experiments using configuration 11 in Table 2 were run for each scheduler configuration. The result box plot is shown in Figure 5. The statistical results are also shown in Table 3. With the results from the repeated experiments, an independent samples t test for $H_1 : \mu_{default} \neq \mu_{gang}$ is formed (where $\mu_{gang}, \mu_{default}$ are the mean job run times using the gang and default scheduler respectively.). The t test shows $p = 0.4$ which fails to reject the null hypothesis ($\alpha = 0.05$).

7 Discussion

Because the gang scheduler always prioritizes allocating resources to one job over another, the first completed job when using the gang scheduler should complete faster than the first completed job when using the default scheduler. But only in similar experiment configurations and if resources are limited to one job at a time. All experiments seen in Figure 3 with three concurrent jobs confirms that the first job completes faster when the gang scheduler is used compared to when the default scheduler is used. If this is true, it would make the gang scheduler more suitable in an online fashion, where jobs are received over time and earlier

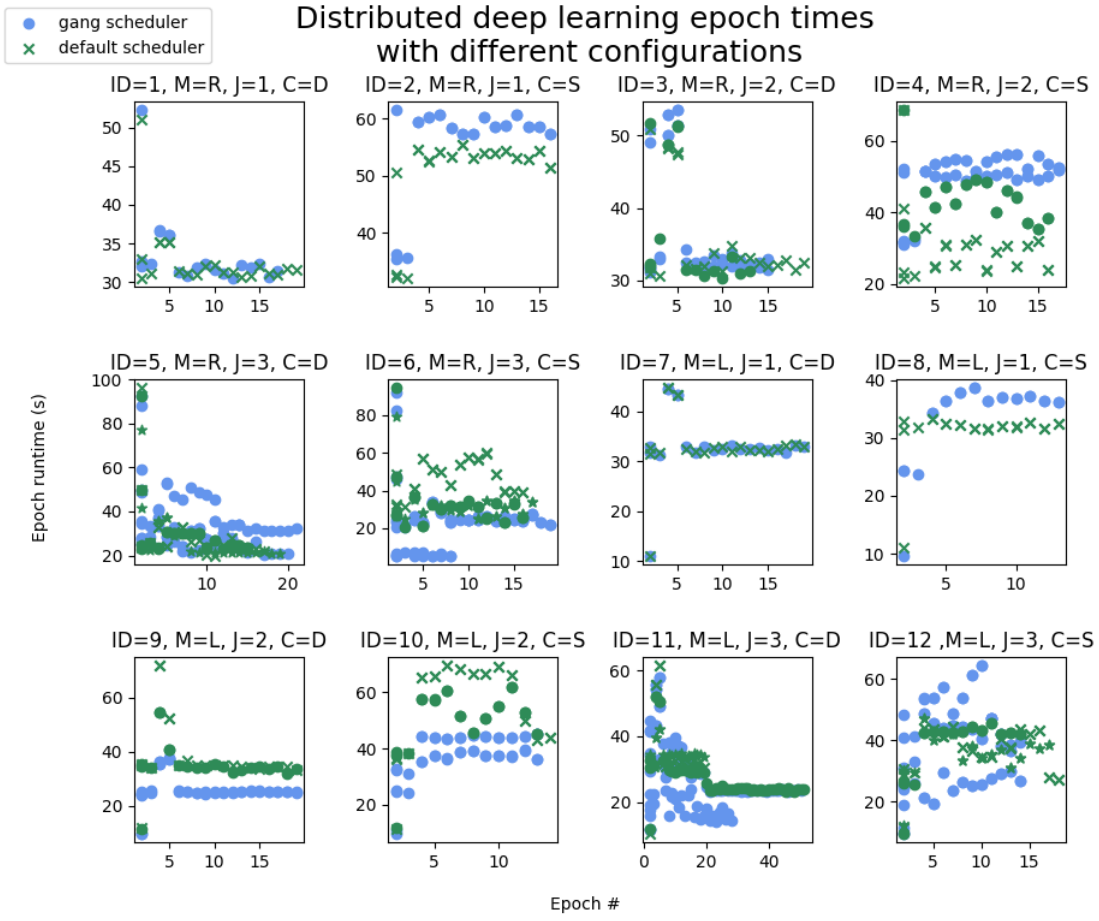

Figure 4: Run-time of the distributed deep learning training epochs and epoch count. Epoch #1 is the first epoch of a job and is incremented by one for each training epoch. Jobs are not differentiated in the plot but scheduler is. Each experiment uses different configurations and schedulers and is managed by a multi-tenant job handler. The configurations include different deep neural network models with datasets, dynamic or static worker and server count and multi- or single-tenant workloads (up to three concurrent jobs). All jobs in the same bar-plot uses the same model and dataset, but uses different schedulers. M is the model and dataset used, J is the job count and C is whether the number of workers and servers are dynamically changed or static. If $C = S$, the static count is set to two for workers and servers. $M \in \{ResNet18, LeNet\}$, $J \in \{1, 2, 3\}$, $C \in \{D, S\}$. All jobs of the same plot are of the same configuration (model, dataset, target loss and budget).

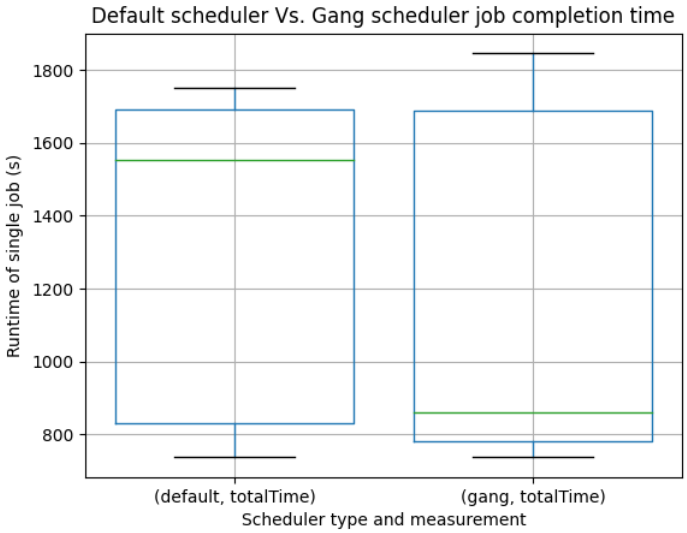

Figure 5: Job completion time when the default and gang schedulers were used for three-job multi tenant experiments with dynamic worker and server selection with the Coach distributed deep learning job handler.

jobs are expected to be allocated resources before later jobs. However, the last job to complete is almost always slower to complete when the gang scheduler is used compared to when the default scheduler is used, as can be seen in Figure 3. Reasonably, when a job completes, fewer jobs are competing for cluster resources which should reduce the scheduling time and improve the performance of the remaining jobs. Perhaps the additional wait time the gang scheduler adds (as described in Section 4.5 results in the slightly slower completion time of the last job compared to the default scheduler.

Because we cannot reject the null hypothesis of the schedulers having equal mean job completion times, it seems there is no significant difference between them. This means that the benefit of reducing the risk of resource deadlocks described in Section 3.2 comes at no additional cost in scheduling time in small scale clusters. Additionally, the gang scheduler prevents scheduling of pods until all types of pods for a single job has been requested for scheduling. This added overhead might be small (because all job types are typically scheduled simultaneously) but is not visible in the mean completion time of the repeated experiments, which hints at the gang scheduler being slightly faster for three multi-tenant experiments.

It is also worth noting that there does not seem to be any performance difference between the two schedulers for the different experiment configurations. That is, it does not seem to matter how many concurrent jobs are active, or if more or less workers are scheduled for each epoch. Hypothetically, the gang scheduler should reduce the mean completion time the more jobs are competing for resources. This is because of both the reduced risk of resource deadlocking, which requires manual intervention, and because of reduced straggler count. Straggler count being parameter server deployments (at least one worker, server and scheduler) with fewer than requested allocated workers or servers. Having too few workers and servers might cause straggler epochs that completes epochs slower than expected because of full memory or bandwidth.

The way Coach runs an experiment is by iteratively performing its major steps as described in Section 4 (initial tuning and training). This step is done with all jobs concurrently before moving to the next step. This means that when a job completes its initial tuning, it must wait

for all other jobs to complete their initial tuning before the *real* training can start. If the gang-scheduler provides faster initial tuning to one job, allowing it to complete its tuning fast, it is still blocked by all other jobs. This might cause a straggler scenario, where resources are not maximized because multiple jobs are waiting for a final straggler job that only utilizes a fragment of all available resources. It could potentially cause a deadlock, because resources are not freed when the initial tuning is finished. Instead of blocking, the multi-tenant implementation should allow jobs to run from the initial tuning step, to the training step without having to wait for other jobs to finish. This is simply a mistake in the implementation of the multi-tenant handling of Coach and might have affected the result of the multi-tenant tests.

7.1 Conclusion

It seems like there is no major difference between the two schedulers on the different experiment setups. Although some experiments showed the gang scheduler to be faster, and some showed the default scheduler to be faster, no clear difference could be seen between the schedulers. With the (potential) exception of the first job being completed faster when the gang scheduler is used with three concurrent jobs. This faster initial completion time would make the gang scheduler more suitable in an online fashion compared to the default scheduler. Because jobs would complete in a queue-like order compared to a more random order when the default scheduler is used. Additionally, because the gang scheduler prevents resource deadlocking, while still having similar completion times to the default scheduler, it is a clear winner for at least a three node cluster with three concurrent jobs. Some results might have been skewed by the job handler implementation. Specifically because the job handler blocks jobs from working until an initial step has completed for all jobs. This could potentially lead to lower completion times for the gang-scheduler, or resource deadlocks.

8 Future work

Because of limited time and computing resources, the results does not fully represent how both schedulers would perform in a large scale cluster used by for example Open AI (with 7500 nodes)[15]. More experiments with additional configurations and larger clusters would add depth and clarity to the results. Not only because the experiments would be more industry representational, but also because of increased variation in resource allocation requests. That is, more variance in the number of deployed workers or servers.

It would also add value to run experiments with larger, more resource intensive workloads/-models. Because a function invocation can finish relatively quickly and free its resources, the problem of resource locking and job straggling is reduced compared to if a job would require longer invocation times and more resources. After an epoch training task is submitted for scheduling, training will not start until one of each type of pod has been scheduled. If pods are too resource hungry, the default scheduler might cause resource deadlocks. For example, if one *server* and one *scheduler* for job *A* and a *worker* and *server* for job *B* are given all available resources, these functions will never be able to start because they are missing one type of function (*worker* for job *A*, and *scheduler* for job *B*.). Bigger workloads in the form of more concurrent jobs is also of interest. Even on small scale clusters. This is because the potential of the gang scheduler increases the more jobs are competing for resources.

Our job handler was initially able to handle being constrained by a budget for each job. If

[15] https://openai.com/blog/scaling-kubernetes-to-7500-nodes/

the cost of training threatened to go over-budget, the job handler would dynamically reduce the number of servers or workers launched per epoch. However, due to time constraints and unintended behavior of the job handler, the budget management was disabled. Budget handling is an essential step of any software-as-a-service distributed deep learning training framework, but should not change the result of the scheduler comparisons. Additionally, some bugs still exists in Coach, and if they are fixed, more results could be run. An example of a potential bug is that some jobs train for double the number of epochs than its multi-tenant neighbors does, as seen in Figure 4 experiment 11. However, these potential bugs should not change the result of the scheduler experiments in any meaningful way.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016.

[2] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[3] Neil Briscoe. Understanding the osi 7-layer model. *PC Network Advisor*, 120(2):13–15, 2000.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[7] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research*, 2018.

[8] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks, 2018.

[9] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[10] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[11] Ying Mao, Yuqi Fu, Wenjia Zheng, Long Cheng, Qingzhi Liu, and Dingwen Tao. Speculative container scheduling for deep learning applications in a kubernetes cluster, 2020.

[12] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Ioan Raicu, Ian Foster, Alexander Szalay, and Gabriela Turcu. Astroportal: A science gateway for large-scale astronomy data analysis. 01 2006.

[14] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021.

[15] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The spec cloud group's research vision on faas and serverless architectures. WoSC '17, page 1–4, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Vinod Vavilapalli, Arun Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: yet another resource negotiator. 10 2013.

[17] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *CoRR*, abs/1912.09789, 2019.

[18] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[19] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019.

[20] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.

[21] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1355–1364, New York, NY, USA, 2015. Association for Computing Machinery.

9 Appendix

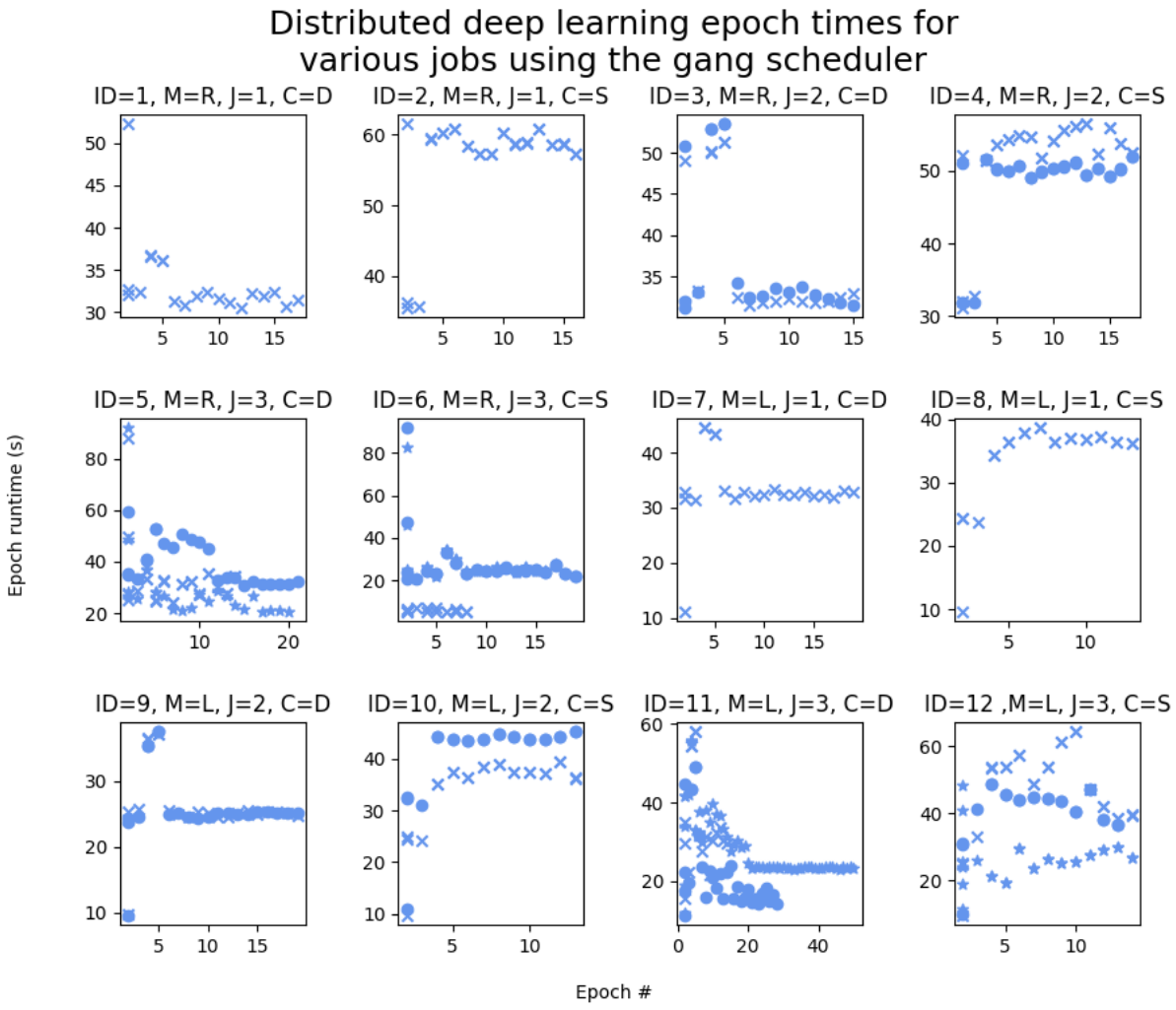

Figure 6: Run-time of the distributed deep learning training epochs and epoch count using the Kubernetes gang scheduler. Epoch #1 is the first epoch of a job and is incremented by one for each training epoch. Different jobs have different markers, (stars, circles and crosses). Each experiment uses different configurations and schedulers and is managed by a multi-tenant job handler. The configurations include different deep neural network models with datasets, dynamic or static worker and server count and multi- or single-tenant workloads (up to three concurrent jobs). All jobs in the same bar-plot uses the same model and dataset. M is the model and dataset used, J is the job count and C is whether the number of workers and servers are dynamically changed or static. If $C = S$, the static count is set to two for workers and servers. $M \in \{ResNet18, LeNet\}$, $J \in \{1, 2, 3\}$, $C \in \{D, S\}$. All jobs of the same plot are of the same configuration (model, dataset, target loss and budget).

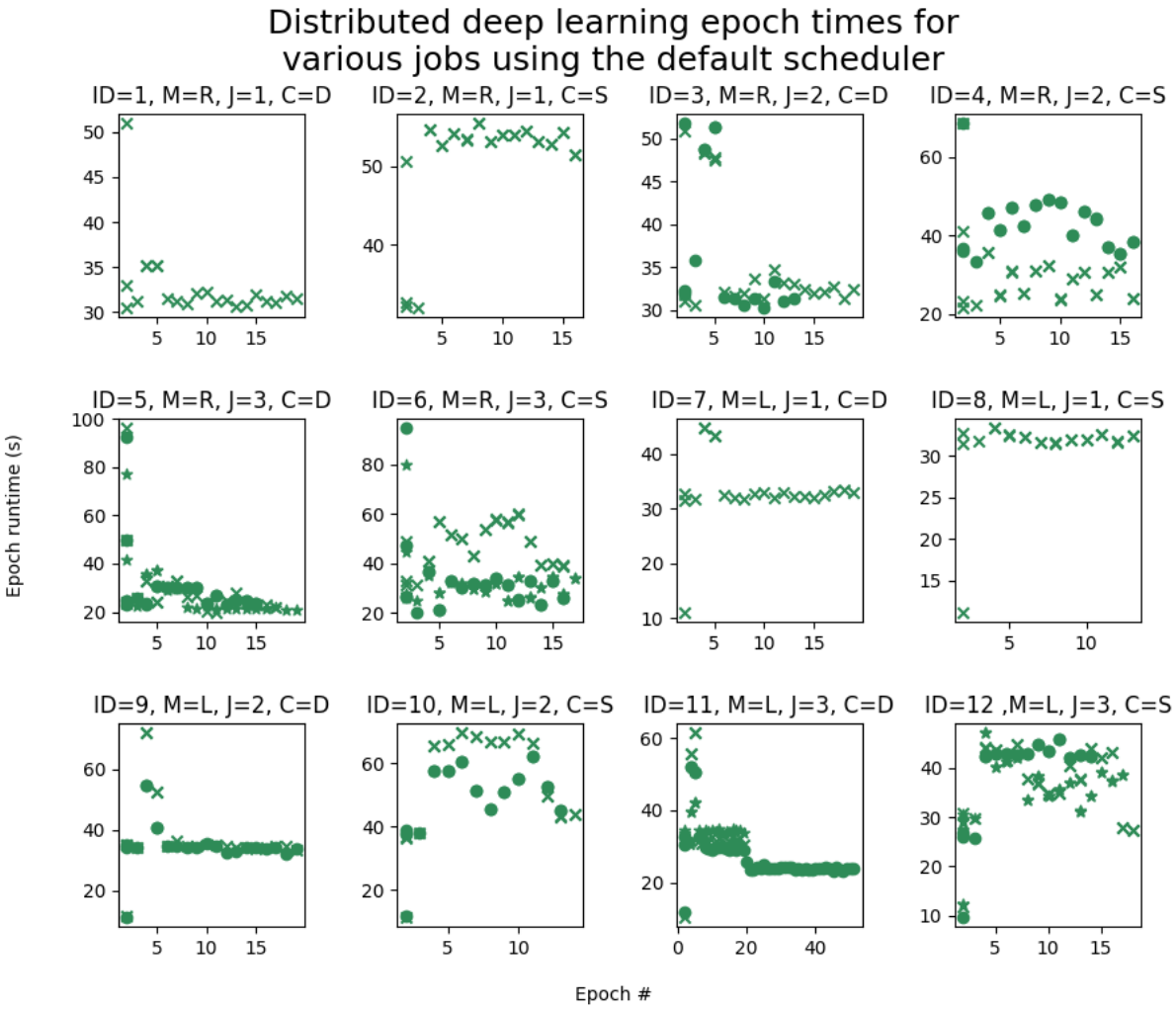

Figure 7: Run-time of the distributed deep learning training epochs and epoch count using the default Kubernetes scheduler. Epoch #1 is the first epoch of a job and is incremented by one for each training epoch. Different jobs have different markers, (stars, circles and crosses). Each experiment uses different configurations and schedulers and is managed by a multi-tenant job handler. The configurations include different deep neural network models with datasets, dynamic or static worker and server count and multi- or single-tenant workloads (up to three concurrent jobs). All jobs in the same bar-plot uses the same model and dataset. M is the model and dataset used, J is the job count and C is whether the number of workers and servers are dynamically changed or static. If $C = S$, the static count is set to two for workers and servers. $M \in \{ResNet18, LeNet\}$, $J \in \{1, 2, 3\}$, $C \in \{D, S\}$. All jobs of the same plot are of the same configuration (model, dataset, target loss and budget).

UMEÅ UNIVERSITY