Efficient Resource Management for Deep Learning Applications with Virtual Containers

BY

Wenjia Zheng

BA, Xiamen University of Technology,

Contents

Acknowledgement

I would like to express my sincere gratitude to my advisor Prof. Ying Mao for the continuous support of my master's study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I have learned many things since I became Prof. Mao's Research Assistant. He spends very much time instructing me on how to become a good researcher and deliver an abundance of knowledge to me.

My sincere thanks also goes to the rest of my thesis committee: Prof. Xiaolan Zhang, and Prof. Mohammad Ruhul Amin, for their encouragement, insightful comments, and hard questions.

Special thanks are given to my fellow labmates: Yuqi Fu, Michael Tynes, Yun Song, Henry Gorelick, for their support. It is the integration of our knowledge and the countless time spent making this research possible. I won't be able to finish this thesis without their help.

List of Tables

List of Figures

INTRODUCTION

The starting of deep learning back to 1943: Walter Pitts and Warren McCulloch created a computational model based on the neural networks of the human brain called threshold logic. After that, deep learning encountered two artificial intelligence winters until late 1990, when GPUs (graphics processing units) were developed and computational abilities of processing data become faster[1]. Besides, data explode, many large open datasets have been launched, such as ImageNet[2], Open Images Dataset[3] and CIFAR-10[4]. As a result, the development of deep learning has been accelerated, and various new architectures of neural networks like AlexNet[5], GoogLeNet[6], ResNet[7] emerged. With the stronger computing power and increasing volume of data, deep learning techniques have improved the accuracy of recognizing, classify, detect, and an increasing number of companies are using deep learning to support their business. For example, Facebook uses deep learning for image detection and object classification, Google and Tesla are involved in developing self-driving cars by deep learning, and Netflix applies deep learning to content recommendations.

A deep learning training job aims to obtain a well-trained model that can get as high accuracy as possible for predictions, and to optimize the model parameters to minimize the loss. To achieve this, a deep learning job trains its model iteration by iteration, applies optimization algorithm and updates a combination of weights according to its loss function. The loss function estimates the loss of the model in each iterations so that the set of weights can be updated to reduce the loss in the next iteration.

Due to a great quantity of training data, a large number of parameters and the number of iterations, deep learning training job relies on computational capabilities — or ability, and it takes a lot of time and resources to train deep learning models before it gets the result. For Instance, training ResNet-50 90 epoch with ImageNet-1k on an NVIDIA M40 GPU takes 14 days. Two weeks of training time takes a long waiting time to get the training result, not to mention training processes usually need to repeat several times to adjust parameters and get the best model. The time cost and resources cost for deep learning training jobs is very expensive. The most common approach to reduce training time is to add hardware resource. Facebook[8] finishes the 90-epoch ImageNet training with ResNet-50 in one hour on 32 CPUs and 256 NVIDIA P100 GPUs (32 DGX-1 stations) with the cost of the whole system is about 4.1 million USD[9]. This approach significantly improved training time, however, deploying the hardware like this is very expensive and it is unnecessary and unaffordable for the general public or small companies. Only one training job requires a large number of resources, let alone multiple jobs that are very common in the industry environment. Consider this circumstance, the cloud seems to be a better option to improve the hardware condition. Many cloud computing service providers such as Microsoft and AWS provide a large scale of sharing computing resources, and users are able to create traditional virtual machines or/and virtual containers and run deep learning training jobs inside the virtual machines. In the cloud environment, various jobs are able to run concurrently and compete for computing resources like CPUs and Memories.

To efficiently allocate computing resources, scheduling algorithms are essential for systems. There are some commonly used schedulers such as Docker swarm and Kubernetes[10] which provide service discovery, load balancing, and other container and cluster management services. These general platforms for resource management can balance the workload across containers in the cloud in general, however, they are not designed specifically for deep learning jobs and fail to take into account the characteristics of the training process. For a deep learning training task, the goal is to reduce the loss iteratively until it converges to a minimum number. The converging speed is not static, on the contrary, it decreases generally with time. In the beginning of training, the loss reduces significantly. After a rapid decline phase, the loss will converge to a relatively stable number and the efficiency of the training process decreased. It turns out that although the resource usage stays the same for each unit of the training process, the gain varies over time. While the computing resources usage remains the same, the reduction of loss is very slow after the training converged, and this is a waste of computing resources. Current resource management systems do not adjust the resource allocation particularly in response to this feature of deep learning training application, thus the gains in loss compared to the resources it uses are inefficient.

In this thesis, we focus on large deep learning tasks running in the cloud environment. More specifically, we concentrate on containerized deep learning applications and containerized cloud architectures with the setting that all training tasks are running inside containers. A container is a lightweight, operating system-level virtual machine packaging up code, runtime, system tools, system libraries, and settings. Docker[11] is a commonly used tool to create, deploy and run containerized applications. Many popular deep learning frameworks and libraries like Keras, Tensorflow and Pytorch publish their images for Docker so that developers can easily build their deep learning applications on the top of these images. Since the container is isolated from each other, it is easy for developers to check the status of each running task. For the resource-intensive problem, we consider addressing it from the following two aspects:

- Local optimization: real-time resources configurations on a single machine inside a cluster.
- Global optimization: cluster-wide resource configuration in a cluster of containers.

we have proposed solutions for these two subproblems respectively. We first propose FlowCon, which targets to improve the efficiency of multiple deep learning training applications running on the containerized cloud environment and accelerate the overall makespan for the system by real-time resource allocation on a single machine. Different from current systems that have a fixed configuration, FlowCon is a real-time resource allocation system that monitors resource usage and the progress of training, categories training jobs into different phases based on growth efficiency, and dynamically allocates resources to each job according to its phases for each node in the cluster.

To improve resource allocation through out a cluster, we propose a second container workflow scheduler for deep learning applications called SpeCon, a Speculative Container scheduler, which aims to accelerate multiple running tasks in a cloud cluster. Besides a fixed or static configuration in the worker node in the cluster, the existing schedulers terminate or migrate a container when a the worker node is overwhelmed. This will train the deep learning model from the beginning which will significantly extend the overall system makespan. Similar to FlowCon, SpeCon categorizes a training job into different phases, and decides whether a job converged or not. For a converged job, SpeCon first gathers system-wide resource usages information as well as existing jobs on each node, and then select the most desirable host for migration by using our proposed execute algorithms. The container reallocation in SpeCon releases resources to fast-growing jobs, and therefore improve overall system performance. When all training jobs have converged, SpeCon keeps monitoring the system to rebalance the workloads.

The main contributions of this thesis are summarized as follows:

- To optimize computing cluster resource utilization, we divide it into two subproblems: resource management in a single machine environment and resource management in a cluster-wide environment, and propose two efficient resource schedulers, FlowCon [12] and SpeCon, to address these subproblems respectively.
- We analyze characteristics of training processes for various deep learning models and conduct experiments to study the overhead of saving and resuming jobs in a Kubernetes system. In addition, we introduce the concept of growth efficiency, a measure of the magnitude of the change in the loss function with per unit of computing resource.
- FlowCon is designed to elastically allocate (or deallocate) the resources to (or from) each learning job at running time on a single machine by monitoring growth efficiency of deep

learning jobs, allowing jobs to converge more quickly without significant scheduling overhead.

- We propose SpeCon with a suite of algorithms that monitor the evaluation functions of deep learning models on the worker side and on the manager side, it collects cluster-wide information to calculate a weighted score for each worker to select a most desirable host for the migration. Furthermore, it rebalances workload when all containers are converged.

- We evaluate the two systems in leading container platforms, Docker and Kubernetes, with intensive cloud-based experiments using popular DL frameworks, and we demonstrate the effectiveness of both systems. Specifically, compared to the default system resource allocation scheme, FlowCon can reduce the completion time of individual jobs by up to 42.06% without sacrificing the overall makespan. SpeCon achieves significant improvement in completion time. It reduces the running time of individual jobs up to 41.5% (14.1% on average) and 24.7% for overall makespan.

The remainder of this thesis is organized as follows. In Chapter 2, we discuss the related work. In Chapter 3, we introduce the background and motivation with a representative examples. We present the system architecture in Chapter 4 and the relevant algorithms in Chapter 5. We carry out extensive evaluation in Chapter 6. In Chapter 7, we conclude this thesis.

RELATED WORK

Data-driven applications have been booming our business from various perspectives, such as mobile social networks [13]–[15], edge computing [16]–[18] and cloud-based systems [19]–[23]. As more applications step into the domain, how to learn from the vast amount of data becomes a hotspot in academic.

Fueled by the development of machine learning frameworks and dramatically increased data, deep learning has received an amount of attention from both academe and industry. Facing a large number of training demands of deep learning products, an increasing number of studies are focus on accelerating the training process. In general, increasing computing resources can lead to deep learning training time reduced[24]–[26]. Moreover, if invested an extremely huge amount of resources, the training time would reach the minimum number[27]. This concept, however, does not mean that more resources involved in the training process more efficient. On the contrary, in reality, computing resources are costly and limited, thus, developers or resource providers are seeking well-defined methods to ameliorate the distribution of training tasks.

In recent years, numerous deep learning architectures are developed rapidly, however, most of the improvements are focus on accuracy and quality rather than the efficiency of training processes. A large number of researches proposed advanced deep learning architectures to achieve higher accuracy for results. For instance, Bidirectional Recurrent Neural Networks (BRNN)[28] extends regular Recurrent Neural Network (RNN) to two time directions (forward states and backward states) and splits state neurons to two parts that are responsible respectively for the positive time direction (forward states) and the negative time direction (backward states) so that training can get information from both states simultaneously. This approach is further used in Bidirectional Long Short-Term Memory (BLSTM) neural networks which is the LSTM version of BRNN, and many models[29]–[31] are built on the top of BLSTM. Deep Residual Network (ResNet)[7] explores basic CNN to a deeper architecture and extend the number of layers up to 152 by learning the residual representation functions. This innovation of CNN architecture allows ResNet achieve an high accuracy. While model accuracy has certainly improved, these models are extremely resource-intensive and time-consuming. Training ImageNet dataset with ResNet-50 by using Tesla P100 x8 processor takes 29 hours to achieve 75.3% top-1 validation accuracy[32], and many models need several days, even weeks to get the final result. A powerful hardware system is crucial for supporting those time consuming and resource-intensive novel architectures. Some researchers have recognized these problems and worked on improving resource-intensive problem by adjusting architecture itself. In works[33]–[37], they tried different approaches such as reducing filter size, removing layers, exploiting linear structure to speedup training. Although these works show a good result at a certain point, they are still not commonly used tools in both the academic and industry field. Our systems can apply to any type of deep learning training job, and our goal is to directly optimize training performance without modifying model structures for broader applications.

Deploying deep learning training jobs to the cloud is can have massive computing resources. Different tasks share CPUs and memories of physical machines by utilizing virtual machines, and the cluster servers on the cloud provide more resources to the tasks. On account of the design of virtual machines, however, creating a virtual machine requires a large number of resources, and it also gives rise to provisioning delay[38] since each virtual machine runs its own guest operating system. To address this problem, many works[39], [40] have attempted to reduce the delay, but the provision cost cannot be avoided and is still a waste of virtual machine resources. Another shortcoming of virtual machines is the cost of virtual machine migration. Virtual machine migration is a commonly used technique that migrates virtual instances from one physical machine to another in order to manage the whole system such as load balancing, fault management and low-level system maintenance[41]. However, in virtual machines, hard disk storage is heavy-weight which causes poor performance on virtual machine migration.

Container technology is designed to address these limitations of virtual machines. Based on its design, containers are able to run on a common host operating system while each container is isolated from others. Containers are lighter weight than virtual machines. In addition, the operating time of container migration is much less than that of virtual machine migration[42]. With the repaid development of containers, a lot of systems have been proposed for container scheduling. For instance, [43] proposed an efficient container scheduler, ECSched, which models scheduling problems as a minimum cost flow problem, thus it can make placement decisions with high quality and high speed for concurrent deployment requests. Multiopt[44] is a multi-objective container scheduler that takes five key factors into consideration including CPU usage, memory usage, time consumption, association, clustering, and establishes a composite function by combining scoring functions of each factor. PIVOT[45] introduced a cost-aware scheduler that enables data-intensive tasks to run and scale in the cloud immediately and cost-effectively. Although these novel scheduling algorithms have a good performance in container placement, they are all for the general workload. In contrast, our systems specifically target deep learning applications.

Several previous works focused on container scheduling algorithms for deep learning tasks. TRADL[46] provides a resources optimizer based on user-defined target loss. Once the deep learning training process reached the target loss, TRADL reduces or removes CPU and Memory usages to other training jobs which still need resources to reach the target. The system is well designed to monitor the training progress and resource usages, and it can allocate the resource in real-time. However, TRADL requires a user to predefined the target number. Gandiva[47] uses the predictability of jobs to efficiently time slice GPUs, hence achieve low latency, self-check job performance and dynamically migrate jobs to better adapt to the GPUs and improve cluster efficiency. However, it does not consider performance of deep learning training progress.

ProCon[48] has a limitation of readjustment containerized deep learning applications since it does not support container migration.

Our solution especially targets resource management for containerized deep learning applications in the cloud environment. We aim to optimize resource configuration by considering two settings: (1) computer resource utilization on a single machine, and improve resource allocation between different tasks that sharing the resource on the certain machine/node; (2) focus on global resources through the cluster, use migration technique to reallocate computing resources for running tasks. These settings are implemented respectively through our two efficient resource management system FlowCon and SpeCon.

BACKGROUND AND MOTIVATION

In this chapter, we study the training processes of containerized deep learning applications and motivate our works with experiments.

3.1 Training A Deep Learning Model

Given a deep learning algorithm, model training is the parameter tuning process to approach the global optimum of a predefined evaluation function. Initially, parameters are randomly selected for the model. This model is iteratively fed with mini-batches of training data, where each mini-batch contains features and labels. The model is then evaluated with the evaluation function that informs users how far off does it from the target. The returned value of this function is recorded along with the model parameters. Together, they propagate backward through the neural network architecture. In the next round of the training process, the algorithm updates and adjusts parameters aiming to achieve better performance. Finally, a well-trained model is generated with a parameter combination that produces an optimized evaluation function.

Fig.3.1 presents the training process of three deep learning models, Variational Auto-Encoders (VAE)[49], Bidirectional Recurrent Neural Networks (Bi-RNN)[50] and Gated Recurrent Unit

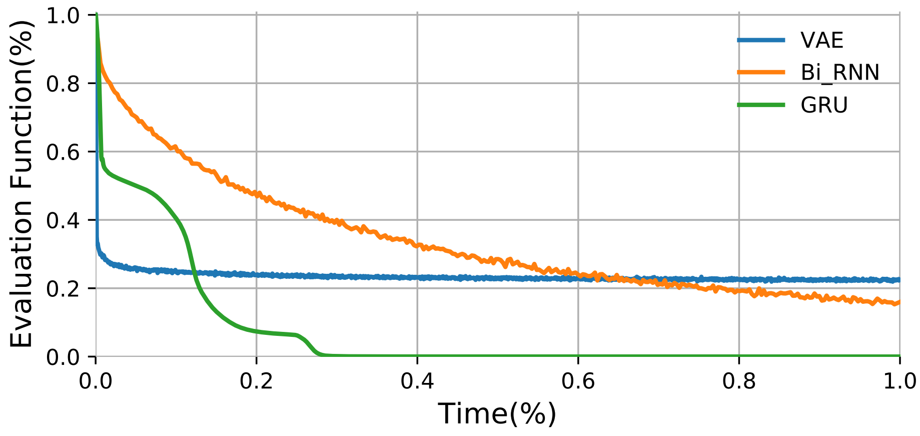


Fig. 3.1: Training progress of three deep learning models

(GRU)[51]. The y-axie is the normalized values of evaluation functions (the lower the better) and the x-axie is the cumulative time. We observe that within the first 20% of the total training time, 65%, 98%, and 90% of the maximum reduction has been achieved for VAE, Bi-RNN and GRU respectively. Since they are trained individually on a physical machine, resources are fully occupied by each model without competition. Therefore, for the same amount of resource, the training gain and efficiency decreases as time goes on.

3.2 Motivation for FlowCon

To deploy applications into a production environment, it is difficult to achieve resilience and scalability using only a single compute node. Generally, a cluster (cloud) is used to provide the infrastructure for running a large set of containers at scale. Many toolkits have been designed for container orchestration in cluster environments, such as the Docker Swarm and Kubernetes.

In current containerized cloud systems, running containers compete for resources freely and the system maintains fairness among all of them on a worker node. Alternatively, users can set an upper limit to each of the containers when initializing them. However, these mechanisms are not optimal for deep learning tasks. There are two main reasons. (1) *Most models don't need to be perfect in a* *distant future, they just have to be good in the near future*. Suppose we have a set of deep learning tasks running on containers within a cluster. In some settings such as real-time data analytics, a model would be frequently requested by applications (e.g., prediction) even *before* convergence is reached. In this case, training the model to an acceptable (rather than perfect) level of accuracy is the most important. (2) *More commonly, some learning tasks converge faster than others, and their models can reach an acceptable state with fewer iterations (i.e. less time)*. If we want to train all models to a *usable* state while minimizing wait time, simply maintaining fairness of all tasks will result in resource waste. This is because jobs that are already in an acceptable state will continue to utilize as much resource as those with many optimizations left to do, even though the nearly-converged jobs only make small gains in optimizing their loss function per unit of computing resource.

Back to Figure 3.1, each model runs inside a container on the same physical node. It can be seen that GRU model training job reaches 0.03% loss at 27% of the time. When it completes training, the loss decrease to 0.02%. This observation indicates with the first 27% of the total time, the model achieves 99% of its minimum. Later, it takes 73% of the total time for another 1% of the accuracy. When there are other learning tasks running in parallel on the same node and seeking computational resources, then it is reasonable to shift parts of the computational resource occupied by GRU training job to other learning tasks. In this work, we propose FlowCon for resource management on a single machine to accomplish this goal.

3.3 Motivation for SpeCon

After single machine/node optimization, we consider cluster-wide resource management since nodes are running inside a cluster. As analyzed above, a learning model, at any given iteration, contains the neural network design (the algorithm itself) and values of the network parameters that it has trained. That is to say, the model progress can be saved during and after training. When necessary, the saved model can be resumed from exactly where they left off.

We experimentally study the overhead of saving and resuming learning models and compare them to the nonstop training process. The experiments are conducted under Tensorflow[52] and Pytorch[53], where the training jobs are running inside a Kubernetes container.

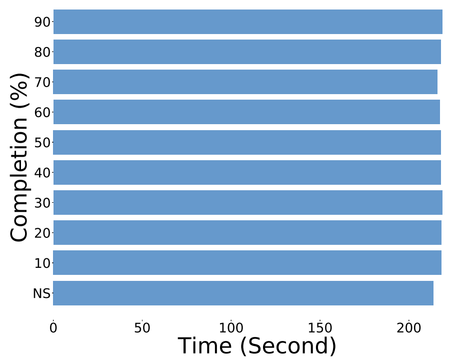


Fig. 3.2: Overhead of saving and resuming VAE at different progress stage

Fig. 3.2 shows the experiments of training a VAE model, where we save the model at a given percentage according to the total number of iterations and resume it immediately after saving completes. Comparing to a nonstop training, denoted as NS on the figure, we find that the overhead is minimum with all less than 5 seconds and consistent with a standard deviation value of 1.43 across different percentages during the training process.

When saving a model in Tensorflow, there are four files generated internally.

- Meta graph file: This is a protocol buffer which saves the complete Tensorflow graph.
- Index file: It stores the list of variable names and shapes saved.
- Data file: This is the file which stores values of all the weights, biases, gradients and all the other variables.
- Checkpoint file: It keeps a record of latest checkpoint files saved. When restoring the model, the system read these files and restart from where it stopped.

Next, we investigate the overhead with different deep learning models. Fig. 3.3 illustrates the comparison between nonstop training and the ones with saving/resuming. The results of completion

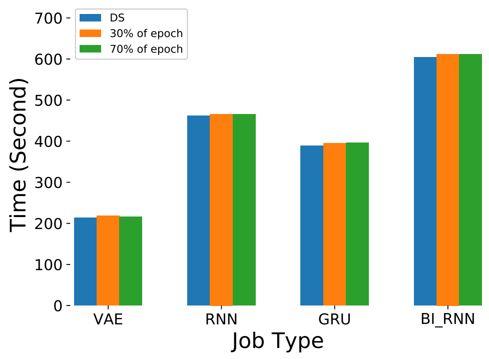


Fig. 3.3: Overhead of saving and resuming deep learning models at 30% and 70%

time are similar as shown on the figure that the overhead is minimum and consistent across different models.

EFFICIENT RESOURCE MANAGEMENT SCHEMES

In this chapter, We introduce two system-level resource schedulers, concerning the features of deep learning applications. These two schemes can improve two subproblems of cluster resource management respectively: resource configuration on a single machine and cluster-wide resource configuration. Section 4.1 presents system design of FlowCon, containerize resource management that target to single machine environment, and Section 4.2 introduces SpeCon, a cluster-wide resource scheduler.

4.1 The FlowCon System

In this section, we present the design of FlowCon in detail, including its architecture, system modules, and the optimization problem it solves.

4.1.1 FlowCon Architecture

In a typical cluster of containers, e.g., Docker Swarm and Kubernetes, there are multiple managers and workers in the system. Managers accept specifications from users and are responsible for reconciling the desired state with the actual cluster state, and workers are responsible for running jobs. Figure 4.1 presents the system architecture of FlowCon. The main components of FlowCon are in the box that represent a worker.

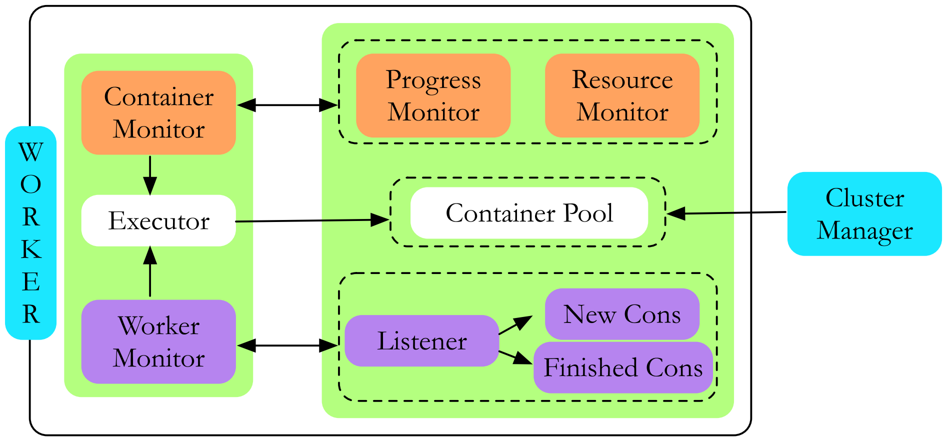


Fig. 4.1: FlowCon System Architecture

Although mangers have all global view of the workers in the system, FlowCon runs on the worker side to prevent overwhelming the manager, which is responsible for collecting the status information from all workers and assigning jobs to them. In our FlowCon system, mangers only interact with the container pools in the workers, which store information of all running containers. With this design, the overhead of running FlowCon is distributed over the entire cluster.

4.1.2 FlowCon Modules

As demonstrated in Figure 4.1, FlowCon consists of three modules, a *Container Monitor*, a *Worker Monitor* and a *Executor*. Each module runs independently and exchanges information about jobs inside the containers as well as the worker status. Their functionality is detailed below.

Container Monitor

The FlowCon focuses on the containers that provide various machine learning services. A container monitor in FlowCon keeps track of the ML/DL jobs inside each container and collects the progress each of the job in terms of different evaluation functions defined by the jobs themselves. Besides that, it collects the resource usage of each container running in the pool. At each container level, it records the consumption of four resources: CPU, memory, block I/O, and network I/O.

Worker Monitor

A worker monitor measures the container pool in the worker. There are two listeners, New Cons, and Finished Cons. Unlike the container monitor, which focuses on the jobs inside a particular container, these two listeners target the status of container pools. The New Cons listener tracks incoming containers and assigns the appropriate resources to them. The Finished Cons listener monitors the containers with finished jobs and and releases their resources to the system.

Executor

The Executor is a key module that collects and analyzes evaluation functions value and resource usage data on the worker. Based on the initial interval, it calculates the required parameters by using the data from Container Monitor and execute an algorithm (described in chapter 5) to update resource configuration for each container. Upon receiving a report from a listener, the Executor will interrupt the current interval, and running the algorithm to update resource assignment based on the new state of the container pool.

4.1.3 FlowCon System Optimization Problem

FlowCon aims to improve resource efficiency, which is generally assumed to be satisfactory in current cloud systems. However, when considering a system with various deep leaning applications, the term ”in use” fails to accurately reflect efficiency, as illustrated in Figure 3.1.

Based on the characteristics of deep learning applications, we introduce a new definition for efficiency based on an application’s evaluation function. Given a system with a set of running containers, $\{c_{id}\}$, each container uses its own evaluation function to assess its machine learning model (e.g., loss reduction and inception score) $E_{c_{id}}(t)$. For each model, based on its $E(t)$, we define the progress score for the container c_{id} below, where $t_i - t_{i-1}$ is the measurement interval, the value $P_{c_{id}}(t_i)$ is called per-second progress within the interval.

$$P_{c_{id}}(t_i) = \frac{|E_{c_{id}}(t_i) - E_{c_{id}}(t_{i-1})|}{t_i - t_{i-1}} \tag{4.1}$$

Here, $P_{c_{id}}(t_i)$ reflects the model training progress over a given time interval. In order to account for the resources used towards the progress, we propose the *growth efficiency* for each container c_{id} with an active deep learning job. $G_{c_{id},r_i}(t_i)$ in Eq. 4.2 represents growth efficiency with respect to different types of resources (e.g., CPU, memory, network I/O and block I/O), denoted by r_i. The denominator $R_{c_{id},r_i}(t_i)$, is a function that returns average resource r_i usage of c_{id} within interval $[t_i - t_{i-1}]$.

$$G_{c_{id},r_i}(t_i) = \frac{P_{c_{id}}(t_i)}{R_{c_{id},r_i}(t_i)} \tag{4.2}$$

FlowCon aims to maximize the sum of growth efficiency for the whole system in each interval, where each learning model has its own evaluation function and can be calculated in real-time. Assume that there are n containers, each runs one job in the system, and $R_{i_{max}}$ denotes the overall resource capacity for r_i. Then, our performance optimization problem can be formalized as $\mathcal{P}$ below, where

G_{c_{id},r_i} can be computed from measurements of $E_{c_{id}}(t_i)$ and $R_{c_{id},r_i}(t_i)$.

$$\mathcal{P}: \text{ Max} \sum_{i}^{n} G_{c_{id},r_i} \tag{4.3}$$
$$\textbf{s.t.} \sum_{i}^{n} r_i R_{imax}$$

4.2 SpeCon System Design

Although FlowCon achieves significant performance improvement in the presence of various deep learning workloads, the proposed system design focuses on a single-machine (or single-node) configuration. In a real industry environment, we need to deploy tasks on the cloud, thus, we further introduce SpeCon to take advantage of the cluster environment and achieve global optimization. SpeCon is implemented on top of Kubernetes, a dominant container-orchestration platform designed to automate the deployment, scaling, and management of containerized applications. In this section, we present the system architecture of SpeCon in detail, including its key modules and their functionalities as well as design logic.

4.2.1 Kubernetes Framework

Generally, a cluster of containers comprises managers and workers. Managers are responsible for system management, such as resource allocation and failure handling. Workers are in charge of hosting containers and executing workloads. In a Kubernetes cluster of containers, a pod is a group of one or more containers which are hosted on the same worker and share the same lifecycle as well as storage resources.

Typically, there are 6 basic units in a Kubernetes cluster as described below: The Kuberlet and Service Proxy reside in workers.

- API Server: It is the main management point of the entire cluster that processes REST operations, validates them, and updates the corresponding objects in storage.
- Controller Manager: It runs controllers, which are the background threads that handle routine tasks.
- Default Scheduler: It watches for newly created Pods that have no node assigned and is responsible for the placement of pods on workers.
- etcd: It is a distributed data storage solution that stores all the data, e.g. configuration, state, and metadata.
- Kuberlet: It is responsible for maintaining a set of pods, which are composed of one or more containers, on a local system.
- Service Proxy: It maintains network rules on nodes, e.g. implementing a form of virtual IP for services.

4.2.2 SpeCon System Architecture

Fig. 4.2 illustrates the SpeCon architecture in a 3-node cluster with one manager and two workers. Since SpeCon is built on top of Kubernetes, Fig. 4.2 includes the above mentioned six components of kubernetes in light blue. The four key modules of SpeCon, **Worker Monitor** and **SpeCon Scheduler** on the manager side, **Container Monitor** on the worker side, **SpeCon Messager** that works in between, as shown in orange,

The four modules are as follows:

- Container Monitor: It runs on the worker side, which provides the resources, such as CPU and memory, for containers to execute the training jobs. It keeps track of the progress of each model from two perspectives: resource usages and current values of its evaluation function.

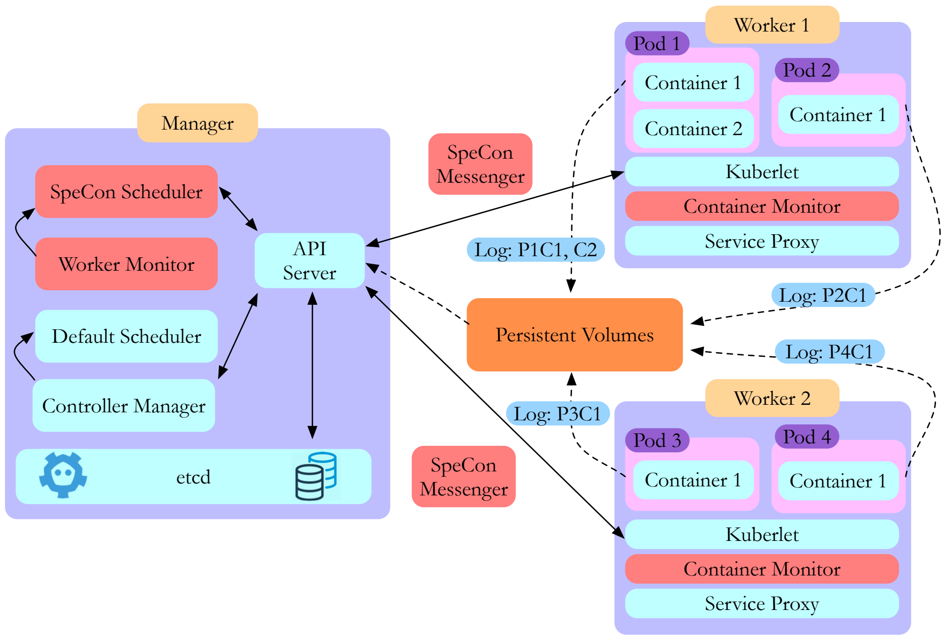


Fig. 4.2: SpeCon System Modules

Based on the collected data, it categorizes training jobs into different stages and activate the corresponding algorithm (details in chapter 5).

- Worker Monitor: It resides on the manager which is responsible for the overall system management. It monitors all workers in the cluster, and keeps track of the number of running jobs and the resource availabilities on each of them. Moreover, it processes messages from workers and communicates with SpeCon scheduler for further actions.

- SpeCon Scheduler: It works on the manager side. When the worker monitor decides that a training job needs to be migrated to another worker node, it use data that gathered by Worker Monitor to execute the algorithm to calculate a score for each of the worker, and then, selects a most desirable node to host this migrated training job (details in chapter 5).

- SpeCon Messager: It generates heartbeat messages to be exchanged between managers and workers. For example, each worker, periodically, sends a message to notify the manager the number of training job hosted on it and their categories. Additionally, whenever a job needs to be migrated, the worker makes use of this channel to notify the manager immediately.

The training jobs in SpeCon write their progress information into the log system in the system. Particularly, SpeCon utilizes the Persistent Volume (PV [54]), which is a piece of storage in the cluster and has a lifecycle independent of any individual Pod that uses the PV, to store the logs. This provides the stability to the SpeCon system.

SOLUTION

5.1 Solution of FlowCon

In this section, we present the design of the elastic container configuration algorithms in FlowCon, which adjusts resource assignment for containers at run time.

5.1.1 Resource Configuration for Containers

In a traditional cluster of virtual machines (VMs), each VM is assigned with a fixed amount of resources (e.g., CPU cores and memory) when the guest operating system is installed. While dedicating VM's for each job enables better isolation, a cluster of VMs fails to efficiently utilize resources for deep learning jobs given their characteristics as we discussed in Chapter 4.

In a cluster of containers, system administrators have the option to create, configure, and reconfigure containers in real time. If the containers are started without a specific resource limit, they will compete for resources at runtime just like processes in an operating system. However, the resource plan can be updated at any time after initialization. For example, the command $docker$ $update < options > container_{id}$ can reset the resource limit as desired. The sample options include –$cpus$ for the number of cores, –cpu-rt-$runtime$ for CPU real-time runtime in microseconds, –$memory$ for memory usage in MB, –$blkio$-$weight$ for a relative weight of block I/O and etc. Finally, limits set by the 'docker update' commands are soft limits, which means that the when the container does not fully utilizes its allocated resource, the unused option will be utilized by others.

5.1.2 Resource Assignment in FlowCon

In a cluster of containers, the manager accepts jobs from users and selects a worker to host the containers. Containers compete for resources such as memory and CPU when they are running in a same worker. By default, each container is assigned the same priority resulting in uniform resource distribution among all containers in the worker. This sharing mechanism yields acceptable performance. However, as we have discussed, it fails to consider characteristics of deep learning applications. In comparison, FlowCon utilizes a growth-efficiency based method as presented in Algorithm 1 to update resource allocation of each active container dynamically.

As shown in Line 1 of Algorithm 1, each W_i first receives the following parameters from its manager: time t, threshold α and algorithm interval $itval$. It then initializes three lists as below:

- New List (NL): Containers that are young and quickly growing
- Watching List (WL): Near convergence containers
- Completing List (CL): Converging and growing slowly

Based on the threshold and the growth efficiency that calculated by the container monitor, the algorithm places each active container into the proper list (Lines 2 - 13). If all containers are in the CL, then each container's resource limit is set to 1 allowing them to compete freely for resources (Lines 14 - 16). While FlowCon is permitting free competition, it is no longer necessary for the system to run the algorithm at the initial interval. Instead, FlowCon utilizes an exponential back-off

Algorithm 1 Dynamic Resource Mgt. for Container c_{id} on Worker W_i

```
1: Initialization: $c_{id} \in \{c_1, c_2, ..., c_n\}$, $W_i$, Time $t$, Watching List $WL$, Completing List $CL$, New List $NL$, $\alpha$ and $itval$.
2: for $c_{id} \in W_i$ do
3:     Calculate $G_{W_i,c_{id}}(t)$
4:     if $G_{W_i,c_{id}}(t) < \alpha$ & $c_{id} \in NL$ then
5:         $NL$.remove($c_{id}$)
6:         $WL$.insert($c_{id}$)
7:     else if $G_{W_i,c_{id}}(t) < \alpha$ & $c_{id} \in WL$ then
8:         $WL$.remove($c_{id}$)
9:         $CL$.insert($c_{id}$)
10:    else if $G_{W_i,c_{id}}(t) \geq \alpha$ then
11:        $NL$.insert($c_{id}$)
12:        $WL$.remove($c_{id}$)
13:        $CL$.remove($c_{id}$)
14: if $\forall c_{id} \in W_i, c_{id} \in CL$ then
15:    for $c_{id} \in W_i, r_i \in R$ do
16:        $L_{c_{id},r_i} = 1$
17:    $itval = itval \times 2$
18: else
19:    for $c_{id} \in W_i, r_i \in R$ do
20:        if $c_{id} \in CL$ then
21:            $L_{c_{id},i} = \frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$
22:            $L_{c_{id},i} = Max\{L_{c_{id},i}, \frac{1}{\beta \times |c_{id}|}\}$
23:        else if $c_{id} \in WL$ then
24:            $L_{c_{id},r_i} = L_{c_{id},i}$
25:        else
26:            $L_{c_{id},i} = \frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$
```

scheme to double the value of $itval$ in order to reduce the overhead of running the algorithm (Line 17). Once the growth efficiency is less than the preset threshold, FlowCon applies the following rules:

- Each container in the CL has its resource limit set based on its growth during the time interval $\frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$ (Lines 18 - 21)
    - If growth is exceedingly small, which is common after convergence, the resource limit is set to a lower bound to prevent abnormal behavior caused by limited resources (Line 22).
- The resource limits of containers in the WL remain unchanged (Line 24).

Algorithm 2 Listener on Worker W_i

```
1: Parameter Initialization: $CL$, $WL$, $NL$, $itval$, $i = 0$
2: $T(i)$ = total number of container at iteration $i$
3: if $i \neq 0$ then
4:     $c = T(i) - T(i-1)$
5: if $c > 0$ then
6:     for $c_{id} \in W_i$ & $c_{id} \notin CL$ & $\notin WL$ & $\notin NL$ do
7:         $NL$.insert($c_{id}$)
8:     $itval$ = initial_value
9:     Run Algorithm 1 and $i++$
10: else if $c < 0$ then
11:     for $c_{id} \in CL \mid \in WL \mid \in NL$ and $c_{id} \notin W_i$ do
12:         $NL$.remove($c_{id}$)
13:         $WL$.remove($c_{id}$)
14:         $CL$.remove($c_{id}$)
15:         Release_resource $c_{id}$
16:     $itval$ = initial_value
17:     Run Algorithm 1 and $i++$
```

- Allocate more resources to containers in the NL (Line 26).

5.1.3 Listeners in FlowCon

The container monitor provides information that allows Algorithm 1 to dynamically allocate resources based on growth-efficiency of each container, and reduce scheduling overhead with an exponential back-off scheme. However, there is latency between the time that a worker's state changes (e.g., a new container is initiated) and the point that it can reallocate resources. To address this issue, FlowCon deploys lightweight background-listeners to track container states in real-time.

With the same set of parameters, Algorithm 2 presents the workflow of listeners on W_i. First, it initializes the CL, WL, NL and $itval$, and uses i to record the number of iteration of the listener (Line 1). When the $i-th$ iteration is running, it uses the function $T(i)$ to fetch the total number of container on the W_i (Line 2). In all runs after the first run, the listener calculates the difference c, between the most recent two iterations (Lines 3 - 4). If $c > 0$, it means that there are c new containers now active in the system, so the listener will stop and the algorithm finds out the c_{id} of the new containers and add them to the NL (Lines 5 - 7). In the meantime, it resets the $itval$ to the original value in order to break the exponential back-off scheme, and then starts to run Algorithm 1 to update the resource allocation as well as increases the iteration number i. (Lines 8 - 9). The case when $c < 0$ indicates that some containers have completed their jobs. The algorithm will then find the relevant containers by their c_{id}, remove them from their associated category (NL, CL or WL) and release their resources (Lines 10 - 15). Finally, we reset the $itval$, start running Algorithm 1 and increment the iteration number i.

5.2 SpeCon Solution Design

As mentioned in chapter 4, SpeCon has a global control of computing resources in a cluster rather than focusing on a single machine. SpeCon aims to improve the system performance by migrating slow-growing jobs to other workers to release resources for the fast-growing ones, by (1) identify slow-growing jobs; (2) selecting the most desirable host; (3) and rebalanceing the workers. The Table 5.1 summarizes the parameters and functions that are used in SpeCon.

Table 5.1: Notations

$w_i \in W$	The worker ID in the worker set
$c_i \in C$	The container ID in the container set
$E_{w_i,c_j}(t)$	The value of evaluation function of c_j on w_i at time t
$G_{w_i,c_j}(t)$	The growth of evaluation function of c_j on w_i at time t
$R(w_i, t)$	Resource consumption of w_i at time t
PC	The progressing category set
WC	The watching category set
CC	The converged category set
α	Categorization threshold
$t_m - t_{m-1}$	Categorization interval
T	Candidate set
S_{w_i}	The scoring set that stores weighted scores for each worker
bf	The balance factor for an uniform distribution

5.2.1 Identifying Slow-Growing Jobs

In SpeCon, we consider a cluster of containers that are hosted on workers. Inside the cluster, we have managers (denoted as M). In addition, we denote $c_i \in C$ to be the set of containers and $w_i \in W$ to be the set of workers. We use $c_m \in w_n$ to denote that a container, c_m, is running on a worker w_n.

In our problem setting, training jobs are running inside containers, where each container hosts one specific job. Consequently, each container, c_i, can be seen as one particular training job in our setting. As analyzed in the previous sections, each job has a predefined evaluation function. During the whole training process, the value of the function forms a time series. When queried in the middle of an iteration, the previous value will be returned. Based on the query time, Equation 5.1 presents the growth of the training job in a given interval, $t_2 - t_1$.

$$G_{w_i,c_j}(t_2) = E_{w_i,c_j}(t_2) - E_{w_i,c_j}(t_1), \tag{5.1}$$

where c_j is a container that runs a training job on w_i with $E_{w_i,c_j}(t)$ as the evaluation function.

According to $G_{w_i,c_j}(t)$, SpeCon classifies c_i into 3 different categories.

- Progressing Containers (PC): The jobs inside these containers are still in the fast growing stage such that their evaluation function is progressing quickly.
- Watching Containers (WC): When a training job join WC category, it indicates that this job is slowing down on the gain of the evaluation function. In this zone, the gain starts jittering and, depending on the model particular, it may speedup the progress again.
- Converged Containers (CC): If training jobs in WC continue slowing down, they are inserted into a Converged Container category, which suggests that they are slow-growing on their evaluation functions and unlikely to speedup again.

Based on the above analysis, we develop the Algorithm 3 in SpeCon to keep tracking the progress

Algorithm 3 Container Monitor: c_j on w_i

```
1:  System Parameters:
        Categorization Interval : $t_m - t_{m-1}$
        Categorization Threshold : $\alpha$
        Evaluation Function : $E_{w_i,c_j}(t)$
2:  Initial Settings:
        $\forall c_{id} \in w_i, PC_{w_i}$.insert$(c_{id})$
        $\forall c_{id} \in w_i, c_{id.\text{migrated}} = $ False

3:  for $c_j \in w_i$ & $c_{j.\text{migrated}} = $ False do
4:      At time $t_m$: Calculate $E_{c_j,w_i}(t_m)$
5:      $G_{w_i,c_j}(t_m) = |E_{w_i,c_j}(t_m) - E_{w_i,c_j}(t_{m-1})|$
6:      if $G_{w_i,c_j}(t_m) < G_{w_i,c_j}(t_{m-1})$ & $Gw_i, c_j(t_m) < \alpha$ then
7:          if $c_j \in PC_{w_i}$ & $c_j \notin WC_{w_i}$ & $c_j \notin CC_{w_i}$ then
8:              $PC_{w_i}$.remove$(c_j)$
9:              $WC_{w_i}$.insert$(c_j)$
10:         if $c_j \notin PC_{w_i}$ & $c_j \in WC_{w_i}$ & $c_j \notin CC_{w_i}$ then
11:             $PC_{w_i}$.remove$(c_j)$
12:             $WC_{w_i}$.insert$(c_j)$
13:         if $c_j \notin PC_{w_i}$ & $c_j \notin WC_{w_i}$ & $c_j \in CC_{w_i}$ then
14:             $c_j$remains in $CC_{w_i}$
15:             if $|PC| + |WC| > 1$ then
16:                 SpeCon-Messager$(w_i, c_j)$.Request-Reallocation
17:     else if $G_{w_i,c_j}(t_m) \geq G_{w_i,c_j}(t_{m-1})$ & $Gw_i, c_j(t_m) < \alpha$ then
18:         $c_j$ stays in the current category
19:     else if $Gw_i, c_j(t_m) > \alpha$ then
20:         $PC_{w_i}$.insert$(c_j)$
21:         $WC_{w_i}$.remove$(c_j)$
22:         $CC_{w_i}$.remove$(c_j)$
```

of training jobs on each worker. Note that, in order to avoid unnecessary network traffic and maintain scalability, the containers in SpeCon will be reallocated at most once.

As shown in Line 1, Algorithm 3 on each worker first fetches key parameters from the system. The categorization interval ($t_m - t_{m-1}$) determines how frequently a worker checks containers' log files, which store return values of evaluation functions. The categorization threshold, α, is a percentage value that SpeCon uses to decide whether the progress is growing quickly. Then, in Line 2, the algorithm initializes parameters such that all containers are inserted into PC when they join a worker and their migration indicator is set to false at the beginning.

At time t_m, it reads the current value of the evaluation function and calculates the gain during the previous categorization interval, $t_m - t_{m-1}$ (Line 3-5). If the gain is less than the previous round and the threshold α, SpeCon updates the category of this container based on the following condition (Line 6-16).

- If the container is currently in the category PC, but not WC and CC, it will be removed from PC and inserted to WC. It means that this job starts slowing down, but not stable yet.
- If the container is currently in the category WC, but not PC and CC, it will be removed from WC and inserted to CC. It indicates that this job has started its convergence process.
- If the container is currently in the category CC, but not PC and WC, it suggests that the value of the gain during the interval continues decreasing and it has converged. In this condition, it will remain in the CC category and the worker will call SpeCon messenger to send a migrated request of (c_j, w_i) to the manager.

In a scenario that the current gain is less than α, but larger than the previous gain value, this container stays in the current category. This is due to the fact that when a training job goes on, the model randomly selects and updates parameters at each iteration that leads to an unstable trends (bouncing growth values). In this stage, the container remains in the same category and waiting for next round (Line 17-18).

At the moment that the gain becomes larger than the threshold, we reset the container's category to PC, which is utilized to accommodate a sudden change of the evaluation function and possible errors from the previous rounds (Line 19-22).

5.2.2 Scheduling on Slow Moving Jobs

The container monitor that runs each worker keeps tracking the evaluation functions and collects data that stores in the persistent volume. Whenever receives a request from workers, the manager

Algorithm 4 Speculative Scheduling of c_j in w_i on Manager

```
1:  Parameter Initialization:
      Candidate_Set : $T = \phi$
      Weights : $w_{pc}, w_{wc}, w_{cc}$
      Resource Consumption on $w_i$ at time $t$ : $R(w_i, t)$

2:  for $w_j \in W$ do
3:    $S_{w_j} = |PC_{w_j}| \times w_{pc} + |WC_{w_j}| \times w_{wc} + |CC_{w_j}| \times w_{cc}$
4:  for $w_j \in W$ do
5:    Find Min($S$)
6:    if $S_{w_j} =$ Min($S_{w_j \in W}$) then
7:      $T$.insert($w_j$)
8:  if $w_i \in T$ then
9:    $c_{id.\text{migrated}} =$ True
10:   $c_j$ remains on $w_i$
11: else
12:   for $w_j \in T$ do
13:     if $|T| = 1$ then
14:       Save $c_j$ on $w_i$
15:       $c_{id.\text{migrated}} =$ True
16:       Restore $c_j$ on $w_j$;
17:     else if $|T| > 1$ then
18:       Rank $w_j \in T$ with $R(w_j, t)$
19:       Save $c_j$
20:       $c_{id.\text{migrated}} =$ True
21:       Restore $c_j$ on $w_j$ with lowest $R(w_j, t)$;
```

responses the reallocation message, e.g. (w_i, c_j), by executing the Algorithm 4. The main objective of Algorithm 4 is to select the most desirable worker node to host this container, e.g. c_j.

SpeCon utilizes a weighted scoring algorithm to rank worker nodes. As shown in Line 1 of Algorithm 4, it first initializes parameters, which include a candidate set (T) that uses to store targeted workers and it is set to empty initially. Additionally, SpeCon maintains predefined weights for each category as well as resource consumption functions.

For each worker node in the cluster, it calculates a score based on the number of containers in each category and its weights. SpeCon aims to improve the efficiency by allocating more resources to fast-growing jobs and limit resources for slow-growing ones. With this objective in mind, the values of weights have a relationship of $w_{pc} > w_{wc} > w_{cc}$, where priorities are given to progressing containers

(Line 2 - 3).

Given the scores of the workers, the algorithm finds out workers that have the minimum score. Those workers are inserted into the candidate set. If multiple workers have the minimum score, it results in more than one candidates (Line 4-7). The algorithm checks whether w_i, the current host of c_j, is in the set or not. Then the following logic will be executed.

- If w_i in candidate set, worker w_i will be returned, which means that container c_j should continue running on w_i. In this case, the algorithm marks the container as migrated (Line 8-10). This manner reduces unnecessary overhead caused by migration.
- If w_i is not in the candidate set, the algorithm takes two branches. (1) If there is only 1 worker in T, SpeCon select this worker for migration. (2) If $|T| > 1$, SpeCon ranks candidates with their resource usages and selects the one with least resource usage as the new host (Line 11-21).

5.2.3 Rebalancing Workloads in the Cluster

Together with Algorithm 3 and 4, SpeCon distributes the workload based on real-time returns from evaluation functions of containers as well as current resource consumption on workers. At this stage, however, it could make inaccurate decisions due to missing information of finishing time. For instance, if $\forall c_i \in W, c_{i.\text{migrated}} = \text{True}$ that means all jobs have converged, therefore, the workload distribution in the cluster is fixed, which would result in an imbalanced cluster. As intuitive example, when all 4 jobs become converged in a 2-node cluster, W_1 hosts c_1 and c_2 and W_2 contains c_3 and c_4. In a scenario that c_1 and c_2 completes before c_3 and c_4, W_1 runs without any workloads.

In a real cloud environment, it is challenge and costly to obtain an accurate finish time due to the various implementations, dynamic workloads and resource competition. In our solution, SpeCon incorporates the converged duration, which is time length between when a container is marked as converged and current time to compare active containers and re-distribute the workload in a cluster by using Algorithm 5.

Algorithm 5 Rebalance Active Container in the Cluster

```
1: Parameter Initialization:
     Candidate_Set : $T = \phi$
     Resource Consumption on $w_i$ at time $t$: $R(w_i, t)$
     $|w_i|$: The number of active containers on $w_i$
     $t_{c_j}$: The timestamp when $c_j$ is marked as migrated
     $t$, $D$: The current timestamp, duration set $D$
     $bf$, $|W|$: The balance factor, total number of workers

2: for $w_i \in W$ do
3:     $sum = sum + |w_i|$
4:     for $c_j \in w_i$ do
5:         $d_{c_j} = t - t_{c_j}$
6:         $D$.insert($d_{c_j}$)
7:     if $|w_i| = 0$ then
8:         $T$.insert($w_i$)
9: $bf = \lfloor sum \div |W| \rfloor$
10: if $|T| > 0$ then
11:     for $w_i \in T$ do
12:         if $|w_i| < bf$ then
13:             Find $c_j$ with Min($d_{c_j}$) in $D$
14:             $D$.remove($d_{c_j}$)
15:             Migrate $c_j$ to $w_i$
16:             $c_j$.rebalanced = True
17:         else
18:             Continue;
19: else if $|T| = 0$ then
20:     for $w_i \in W$ do
21:         if $|w_i| < bf - 1$ then
22:             $T$.insert($w_i$)
23:     for $w_i \notin T$ do
24:         $\forall c_j \in w_i$
25:             Find $c_j$ with Min($d_{c_j}$) in $D$
26:         for $w_j \in T$ do
27:             if $|w_i| < bf - 1$ then
28:                 Migrate $c_j$ to $w_j$
29:                 $c_j$.rebalanced = True
30:             else
31:                 Continue;
```

The algorithm prepares the required parameters in Line 1. Then, it calculates the sum of active jobs on workers across the cluster and computes the converged duration for each container c_j on each worker w_i. The each converged duration, d_{c_j}, is inserted into a set D (Line 2-6). If there is no active jobs on a particular worker w_i, its id would be inserted to the candidate set T, which stores the workers that can take more workload (Line 7-8). Given tj and $|W|$, the balanced factor, bf, is obtained in Line 9. The value of bf is based on uniform distribution.

Depending on the number of workers in the candidate set T, Algorithm 5 takes the following two branches.

- If T is nonempty, it suggests that, at least, one worker is idling. For each idling worker w_i in T, the algorithm assigns c_j, which has the smallest d_{c_j} to w_i if the number of active containers on it is less than bf and marks c_j to be rebalanced (Line 10-18).
- If T is empty, it means that every worker runs some active containers. Then, SpeCon enumerates all workers and finds the ones that host less jobs than $bf - 1$, which indicates they can hold, at least, one more job. These workers are inserted into T (Line 19-22). Then, the algorithm finds the c_j with the smallest d_{c_j} and, in the meanwhile, c_j runs on a worker w_j such that $w_j \notin T$ (Line 23-25). It basically avoids the scenario that c_j assigns to its current host. With a nonempty T, SpeCon assigns the previously found c_j to w_i which has room for an additional container and marks this container to be rebalanced (Line 26-31).

SYSTEM EVALUATION

In this chapter, we evaluate FlowCon and SpeCon separately with large scale experiments, and present the evaluation results.

6.1 Evaluation of FlowCon

In this section, we evaluate the performance of FlowCon through a set of experiments, carried out in the cloud.

6.1.1 Experimental Framework

FlowCon uses Docker Community Edition (CE) 18.09 and is implemented as a middleware between worker and manager. It receives tasks from the manager, and then directs the given tasks to the worker for execution.

The testbed is built on the NSF Cloudlab [55], which is hosted by the Downtown Data Center - University of Utah. Specifically, the testbed uses the R320 physical node, which contains a Xeon E5-2450 processor and 16GB Memory. To ensure comprehensive evaluation, we test FlowCon with various deep learning models using both the Pytorch and Tensorflow platforms. Table 6.3 lists the models used in the experiments.

Table 6.1: Tested Deep Learning Models (FlowCon)

Model	Eval. Function	Plat.
Variational Autoencoders (VAE) [49]	Reconstruction Loss	P/T
Modified-NIST (MNIST) [56]	Cross Entropy	P/T
Long Short-Term Memory (CFC)	Softmax	T
Long Short-Term Memory (CRF) [57]	Squared Loss	P
Bidirectional-RNN [58]	Softmax	T
Gated Recurrent Unit (GRU) [51]	Quadratic Loss	T

6.1.2 Experiment Setup and Evaluation Metrics

There are two key parameters in FlowCon: (1) α, the threshold for classifying jobs into NL, WL and CL; and (2) $itval$, the interval for running the Algorithm 1. We evaluate the performance of FlowCon with different parameter configurations and compare it with the original Docker system (denoted as **NA** in this subsection) using the following three scenarios:

- Fixed scheduling: the time to launch a job is controlled by the administrator.
- Random scheduling: the launch times are randomized to simulate random submissions of jobs by users in a real cluster.
- Scalability: we evaluate FlowCon with an increased number of learning jobs.

The following three metrics are considered in our experiments.

- Overall makespan: the total length of the schedule for all the jobs in the system.
- Individual job completion time: the completion time of each individual jobs in the system.
- CPU usage: all of our tested deep learning models are computation intensive jobs, we focus on analyzing the CPU usage for better understanding FlowCon.

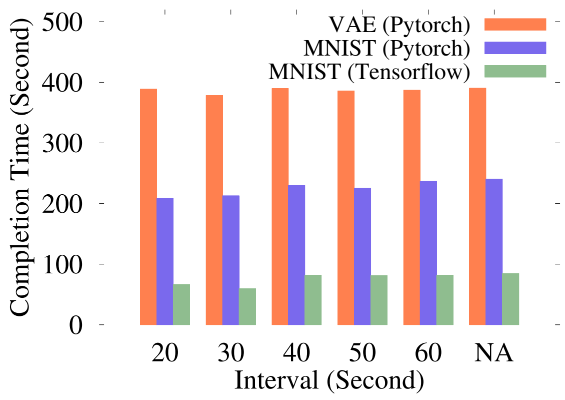


Fig. 6.1: $\alpha = 5\%$ and different values of intervals

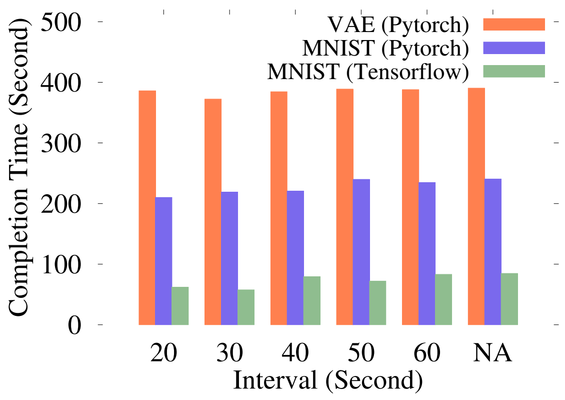


Fig. 6.2: $\alpha = 10\%$ and different values of intervals

Since all the components of FlowCon and the relevant algorithms reside in worker node, in our experiments, we focus on the performance of each individual worker in the system. It should be highlighted here again that the objective of FlowCon is to reduce the individual job's completion time and, at the meanwhile, avoid sacrificing the makespan.

6.1.3 Fixed Scheduling Results

We fix the schedule of three jobs that VAE on Pytorch starts at 0s, MNIST on Pytorch begins at 40s, and MNIST on Tensorflow launches at 80s. We test our system with different input parameters to understand its performance in various settings.

Makespan: Figure 6.1 and Figure 6.2 present results with different $itval$ in the case of $\alpha = 5\%$ and $\alpha = 10\%$ respectively. It can be observed that different $itval$ values have a small effect on the makespan (dominated by VAE), and FlowCon improves makespan by 1% to 5% compared to NA. This is particularly evident when $\alpha = 5\%$, as the makespans are 386.1s, 372.4s, 384.8s, 389.0s, 388.1s and 394.0s respectively. The reason lies in the fact that although FlowCon moves some jobs that have their resource limits constrained, thus slowing them down, the jobs which were allocated more resources finished more quickly, thus reducing the overlap between jobs.

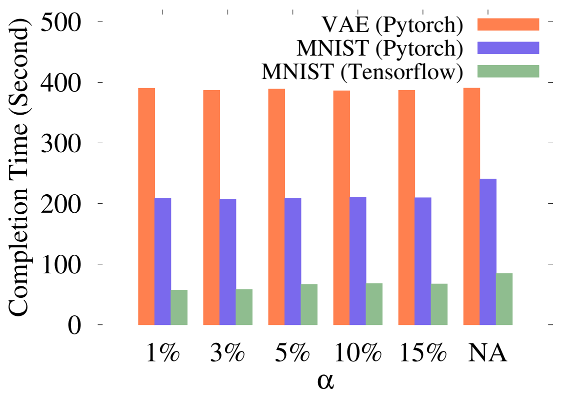


Fig. 6.3: $itval = 20$ and different values of α

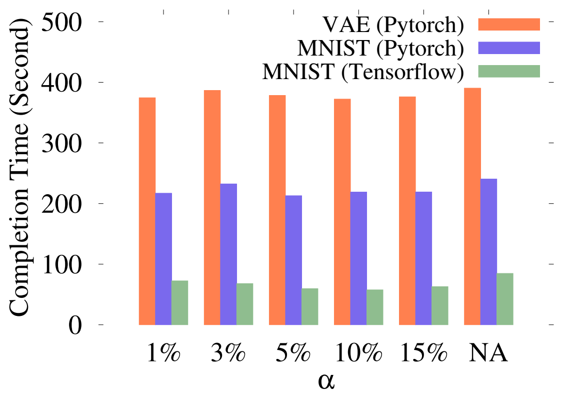


Fig. 6.4: $itval = 30$ and different values of α

Table 6.2: Completion Time Reduction of MNIST (Tensorflow)

$\alpha, itval$ (Fig. 6.2)	Reduction	$\alpha, itval$ (Fig. 6.3)	Reduction
$10\%, 20$	26.2%	$1\%, 20$	32.1%
$10\%, 30$	32.4%	$3\%, 20$	31.0%
$10\%, 40$	14.3%	$5\%, 20$	21.4%
$10\%, 50$	15.3%	$10\%, 20$	19.0%
$10\%, 60$	3.1%	$15\%, 20$	19.8%

In Figure 6.1, when comparing $\alpha = 5\%$, $itval = 20$ with NA, the overlap of the two jobs, VAE (Pytorch) and MNIST (Tensorflow), is 213.2s and 240.5s; while the overlap of three jobs is 59.4s and 84.7s. FlowCon decreases the overlap of three jobs by 29.87%. Due to the decrease of the overlap, the completion time of VAE (Pytorch) (the same value as overall makespan in this experiment) will not increase even if we reallocate its resources to another job at runtime. Figure 6.2 features a key similarity to Figure 6.1. Using a fixed $itval$ and a varied α, as depicted in Figure 6.3 and Figure 6.4, produces similar results as well. Specifically, in Figure 6.3 and Figure 6.4, when $itval = 20$ and $itval = 30$, with different values of α, FlowCon improves the makespan by 1% to 4% across all settings.

Individual: When the completion time of a specific job is investigated, we find a significant improvement in FlowCon. For example, in Figure 6.1, when $\alpha = 5\%$ and $itval = 30$, FlowCon reduces the completion time of MNIST (Tensorflow) by 31.9%, from 84.7s to 57.7s. To show more details about how individual learning jobs can benefit from FlowCon, we extract the results of MNIST (Tensorflow) from Figure 6.2 and Figure 6.3, and present the reduction of its completion time in

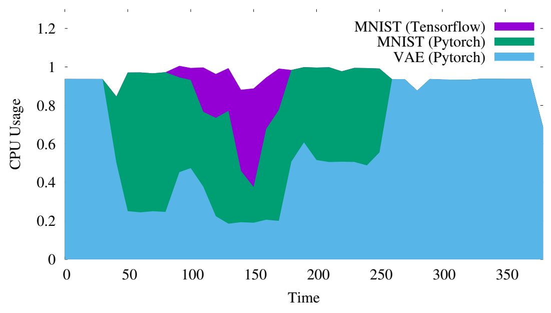


Fig. 6.5: CPU usage of FlowCon ($\alpha = 5\%$, $itval = 20$, 3 jobs)

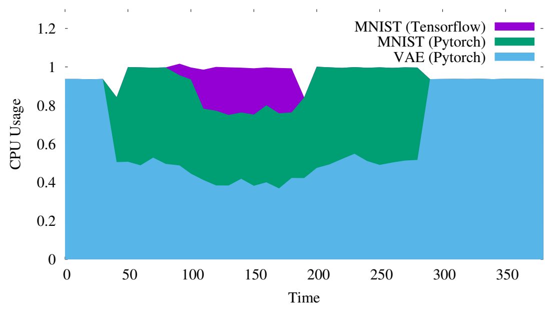


Fig. 6.6: CPU usage of NA (3 jobs)

Table 6.2 by comparing FlowCon with NA. We can see that FlowCon performs better than NA in all the parameter settings. When $\alpha = 10\%$ with $itval = 60$, the performance improvement is the smallest one, only 3.1%. This is because the value of $itval$ is large, and the algorithms need more time to adjust the resource plan for jobs with a large interval. When we fix the value of $itval$ to 20 and vary the value of α, it can be seen that the time reduction generally decreases with the increase of α. The explanation for this result is that jobs stay longer in NL for $\alpha = 1\%$, causing the algorithm to make updates more frequently. For the case $\alpha = 15\%$, jobs stay longer in CL, in which the limits are set to 1, and running tasks will compete for resources freely.

CPU usage: Figure 6.5 and Figure 6.6 illustrate the detailed CPU usage of FlowCon ($\alpha = 5\%$ with $itval = 20$) and NA in the presence of the three jobs, respectively. The results in Figure 6.6 verify that the system equally distributes CPU resources among active jobs when without any configuration (NA). For example, from 40s to 80s and 180s to 280s, the CPU usages of VAE (Pytorch) and MNIST (Pytorch) are approximately equivalent. In comparison, Figure 6.5 shows both that FlowCon can dynamically set the upper resource limit for each job (actually the resource usage also reflects each job's growth-efficiency). Specifically, when MNIST (Pytorch) is launched at time 40s, FlowCon takes two actions: (1) sets VAE's (Pytorch) resource limit to 0.25 since it is growing slowly, and (2) sets MNIST's (Pytorch) resource limit to 1, allowing for a maximum resource. In this case, VAE (Pytorch) will receive 25% while MNIST (Pytorch) will use 75% of the total resources.

6.1.4 Random Scheduling Results

For the random scheduling case, we have used five different deep learning models, LSTM-CFC, VAE, VAET, MNIST, and GRU, in our experiments. We randomly select a starting time point from 0s - 200s to submit a training job, and the responsible jobs are marked as Job-1, Job-2, Job-3, Job-4, and Job-5 respectively in the following results.

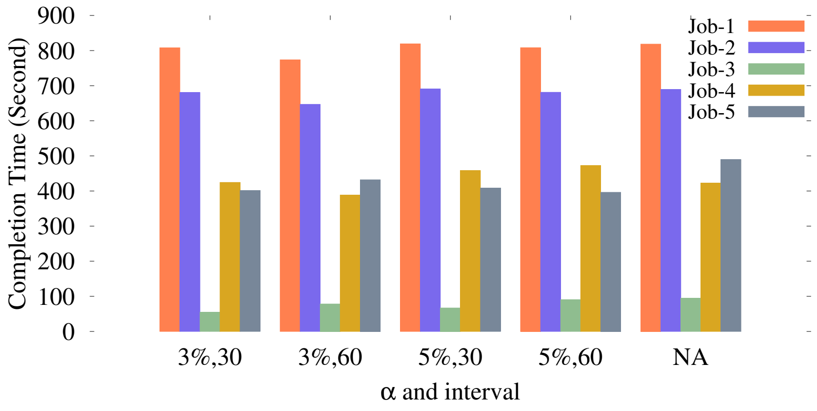


Fig. 6.7: Five different jobs with random submission

Makespan: Figure 6.7 shows the results of system makespan. Similar to the fixed scheduling case, the results here demonstrate, once again, that FlowCon improves the overall makespan, by 1% - 5%. Given the same resource availabilities, FlowCon achieves the reduction of makespan by reducing the overlap between jobs.

Individual: Considering the complete time for each individual job in Figure 6.7, we can observe that FlowCon reduces the completion time for 4 jobs, 5 jobs, 4 jobs, and 4 jobs out of 5 learning jobs for the case with $\alpha = 3\%$, $itval = 30$; $\alpha = 3\%$, $itval = 60$; $\alpha = 5\%$, $itval = 30$; and $\alpha = 5\%$, $itval = 60$, respectively. The biggest loss happens at the fourth job (denoted as Job-4 and others are similar) with $\alpha = 5\%$ and $itval = 60$. There, Job-4 completes in 472.4s, which is 11.80% slower than NA, the completion time of which is 422.5s. The reason is that: although resource allocation to Job-4 greatly decreases when Job-5 begins, the interval of $itval = 60$ prevents FlowCon immediately

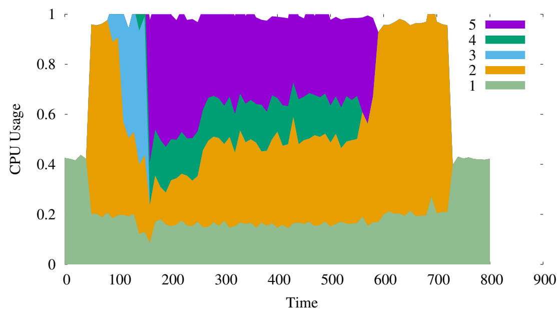


Fig. 6.8: CPU usage of FlowCon ($\alpha = 3\%$, $itval = 30$, 5 jobs)

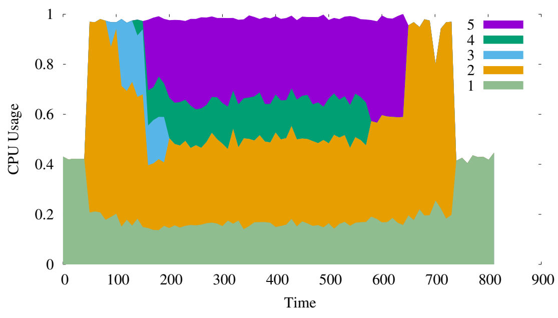


Fig. 6.9: CPU Usage of NA (5 jobs)

reallocating resources from Job-4 to Job-5. However, we can see that FlowCon reduces the completion time of Job-5, by 19.00%, from 489.4s to 396.4s. The largest performance improvement occurs with the setting $\alpha = 3\%$ and $itval = 30$. In this case, Job-3 completes 42.06% faster. Additionally, the makespan of Job-4 increases 2.1s (424.6s vs. 422.5s), and Job-5 completes 17.92% faster (401.6s vs. 489.4s). Therefore, a smaller value of $itval$ will allow FlowCon to reassign the resources more quickly.

CPU usage: The random schedule with a larger number of jobs produces more challenges for resource assignments. Figure 6.8 and Figure 6.9 present the CPU usage of FlowCon with $\alpha = 3\%$ and $itval = 30$ and NA in a system of 5 randomly submitted jobs. Unlike Figure 6.6, the resource usage illustrated in Figure 6.9 is not equally distributed. For example, from 50s to 80s and from 650s to 730s, the first and the second job are active and use 19% and 79% of the resources, respectively. The reason is that: although the first job (LSTM-CFC) is running alone, it does not maximize the CPU usage (e.g., from 0s to 50s as in Figure 6.9). Generally, when a container cannot maximize its resource limit, a portion of the resources may be wasted depending on other jobs in the system. FlowCon addresses this issue with two techniques: (1) sets a soft resource limit to containers. If a container cannot reach its upper limit, then the resources can be used by others; and (2) although the limits of active containers are correlated and based on the growth-efficiency, the sum of all limits can be greater than 1, since containers in CL employ a lower bound: $\frac{1}{\beta \times |c_{id}|}$ (Line 22 in Algorithm 1).

Remark: Based on the experimental results and analysis in chapter 5.3 and 5.4, we can conclude that FlowCon can improve both the system makespan and the completion time of individual learning jobs with different parameter settings, compared to the original Docker system. In the meantime, the values of α and $itval$ can affect the degree of improvement. Since $itval$ indicates the frequency at which Algorithm 1 runs, it is proportional to the overhead including (1) the algorithm resource usage and (2) the delay for reducing the resources of active jobs. Consequentially, as the frequency of running Algorithm 1 decreases, the room for elastic configuration for the running containers will be reduced. Furthermore, the value of α directs how FlowCon categorizes containers in each iteration of the algorithm. Therefore, the best α setting depends on the number of active containers in the system, the machine (deep) learning model in each container, and the corresponding datasets.

6.1.5 Scalability of FlowCon

In this subsection, we conduct experiments to evaluate the performance of FlowCon at a larger scale.

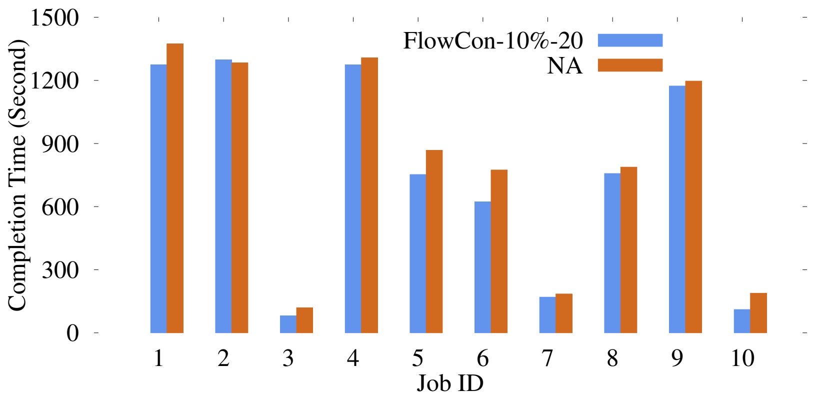


Fig. 6.10: Ten jobs with random submission

10-Job Experiments

Figure 6.10 shows the completion time comparisons of FlowCon with $\alpha = 10\%$, $itval = 20$, and NA with 10 jobs that are randomly submitted from 0s - 200s. Under such settings, the makespans are 1350.7s and 1384.9s for FlowCon and NA, respectively. This follows the same trend demonstrated in previous experiments where FlowCon achieves a small improvement in makespan. Considering jobs individually, FlowCon reduces the completion time of 9 out of 10 jobs. Job-2's completion time increases by 15.4s (1.1%) since the interval between Job-2 and Job-3 is very small (13s), and this quickly leads to resource reduction. However, for the other 9 jobs in the experiment, FlowCon produces makespan reductions from 1.8% to 41.2%, with the largest improvement occurring in Job-10: from 188.3s to 110.8s.

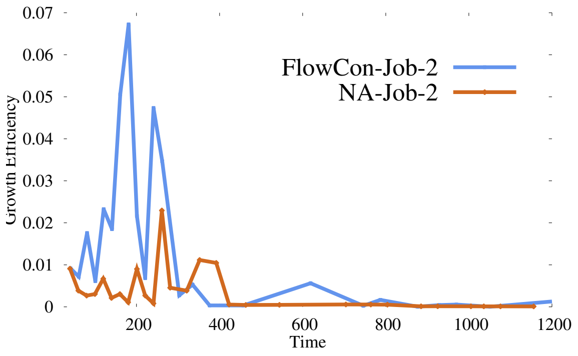


Fig. 6.11: Growth Efficiency Comparison of Job-2

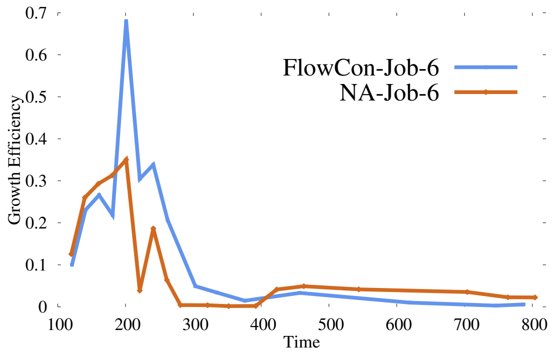


Fig. 6.12: Growth Efficiency Comparison of Job-6

Since the FlowCon updates the container's configuration based on the associated value of growth efficiency, we take a deeper look at the data from the 10-job experiment. We pick up two representative jobs, Job-2 and Job-6, where Job-6 wins and Job-2 losses a bit in terms of completion time. Figure 6.11 and Figure 6.12 illustrate the growth efficiency over the time of Job-2 and Job-6 respectively, in both FlowCon and NA. As we can see from Figure 6.11, FlowCon gains a lot in growth efficiency at the very beginning. The reason lies in the fact that in FlowCon, Job-2 does not need to compete for resources freely due to an upper limit of resources that applies to every job. The more resources allocate to it, the faster it grows. Even when Job-3 joins the system, resources for Job-2 will not reduce too much since it is still in the WL. In comparison, in NA, every job has the same priority and they compete for the resource whenever it becomes available. When Job-2 converges in FlowCon, more resources will be moved to newer jobs due to a smaller value of the limit, which results in a loss when compares to NA at the time point 320s. We find a different trend in Figure 6.12. There, at the first 2 iterations, Job-6, in FlowCon, records sightly lower values of growth efficiency. This is because FlowCon needs more time to update the configuration when there are 5 active jobs in the system. It should be noted that the time for resource usages is shorter than the

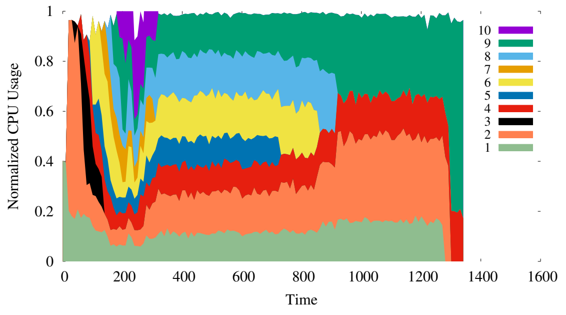


Fig. 6.13: CPU usage of FlowCon ($\alpha = 10\%$, $itval = 20$, 10 jobs)

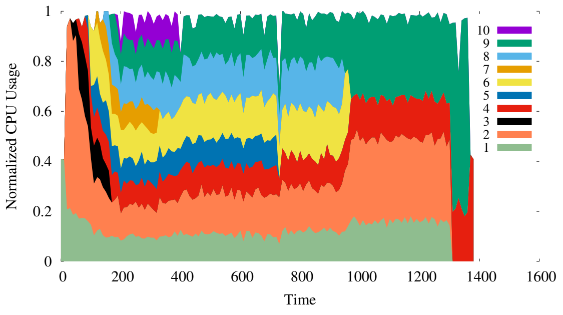


Fig. 6.14: CPU usage of NA (10 jobs)

completion time in Figure 6.10 for a specific container. The difference is caused by our calculation methods. We compute completion time whenever the container is marked as exited and we record the resource usage whenever the Algorithm 1 is called, where the interval could become larger due to the exponential back-off in Algorithm 2.

Figure 6.13 and Figure 6.14 present the CPU usage of FlowCon and NA in the same experiment. From Figure 6.14, we can see the jitter with NA. The jitter is a result of uncontrolled resource competition: whenever there is an idle slot, the system will allocate resources to the first job in the queue. FlowCon also produces jitter, however, the resource usage for each container is much smoother by comparison. This is because FlowCon employs a soft, upper resource limit to the containers, and therefore the room for free competition is reduced. The majority of jitter in FlowCon occurs in the interval between 0s to 200s. In this interval, jobs are submitted to the system randomly. After one container joins the system, resource assignment for each container will be updated to reflect this change in the system's status.

15-Job Experiments

We further increase the number of jobs to 15. Again, jobs are randomly submitted to the system during the interval 0s to 200s. As the number of concurrent jobs increases, so does the degree of competition for resources. The results are presented in Figure 6.15. There, we find the same trend as previous ones: FlowCon reduces makespan from 1980.1s to 1950.9s (1.5%). Comparing with

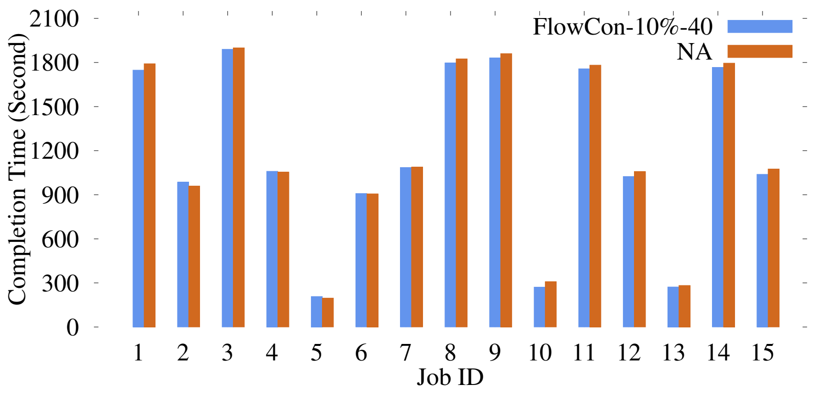


Fig. 6.15: Fifteen jobs with random submission

Figure 6.7 and Figure 6.10 highlights the makespan reduction. The more competition in the system, the greater the challenge for elastic configuration at runtime. Furthermore, when considering individual jobs in Figure 6.15, FlowCon reduces the completion time for 11 jobs out of the 15 jobs. For those 4 jobs, completion time (in seconds) increases: (1) Job-2, from 959.4 to 985.1; (2) Job-4: from 1059.6 to 1053.3; (3) Job-5: from 196.2 to 207.5; and (4) Job-6, from 906.4 to 907.9. However, we can see these increments are quite small, e.g., Job-5's completion time increases the most, only by 5.7%. In comparison, in the other 11 jobs, the degree of reduction ranges from 1.2% to 11.9% and the largest degree of reduction occurs in Job-10, from 308.1s to 271.4s.

6.2 Evaluation of SpeCon

In this section, we evaluate the performance of SpeCon through intensive cloud-executed experiments.

6.2.1 Experimental Setup

Implementation, Testbed and Workloads

SpeCon is integrated into Kubernetes 1.17 as plug-in modules that reside on both managers and workers. It receives tasks from the manager, directs the given tasks to workers, monitors the evaluation functions, and balances the workloads.

We build a testbed on an NSF Cloudlab [55] datacenter that hosts at the University of Utah. Specifically, we use multiple M510 as our physical machines that contain two 8-core Intel Xeon D-1548 at 2.0 GHz, 64GB ECC memory, and 256 GB NVMe flash storage. The following two clusters have been built to evaluate SpeCon.

- **Cluster 1**: 1 Manager and 4 Workers
- **Cluster 2**: 1 Manager and 8 Workers

SpeCon focuses on accelerating the training process of deep learning models. We evaluate it with two popular open-source platforms, Pytorch (P) [53] and Tensorflow (T) [52]. As shown in Table 6.3, we choose 5 different models on those two platforms as our workloads that execute inside containers.

Table 6.3: Tested Deep Learning Models (SpeCon)

Model	Loss Function	Platform
Variational Autoencoders (VAE)	Recon. Loss	P/T
Gated Recurrent Unit (GRU)	Quadratic Loss	P/T
Bidirectional-RNN	Softmax	P/T
Recurrent Network (RNN)	Softmax	P/T
Dynamic RNN	Softmax	P/T

Evaluation Metrics

The training processes of deep learning applications are computationally intensive. They are more sensitive to CPU powers than memory spaces and network bandwidth. The following evaluation metrics are considered in our experiments.

- **Completion time**: The completion time of each job in the cluster.
- **Average completion time**: The average completion time among the jobs in the cluster.
- **Makespan**: The total length of the schedule for all the jobs in the cluster.

When multiple models are training on the same worker, the jobs create a highly dynamic computing environment in terms of resource competition. Therefore, we design the following submission schedules to ensure a comprehensive evaluation.

- **Fixed Schedule**: The time to launch containers follows a fixed interval. It simulates an administrator controlled cloud environment.
- **Random Schedule**: The time to launch a job is randomly selected within an interval. It simulates a user-specified, first-come-first-serve cloud environment.

As for the weight of each category that utilized by Algorithm 4, we use $w_{pc} = 2, w_{wc} = 1.5$, and $w_{cc} = 1$. Additionally, we compare SpeCon with the default scheduler in Kubernetes, which noted as DS in the rest of the evaluation subsection.

6.2.2 Fixed Schedule

Firstly, we conduct experiments with a fixed schedule such that containers with the same model image are submitted to the system one by one with a fixed interval. In this experiment, we use VAE model and 50 second interval. Furthermore, we set $\alpha = 0.01, |t_m - t_{m-1}| = 30$ for SpeCon algorithms.

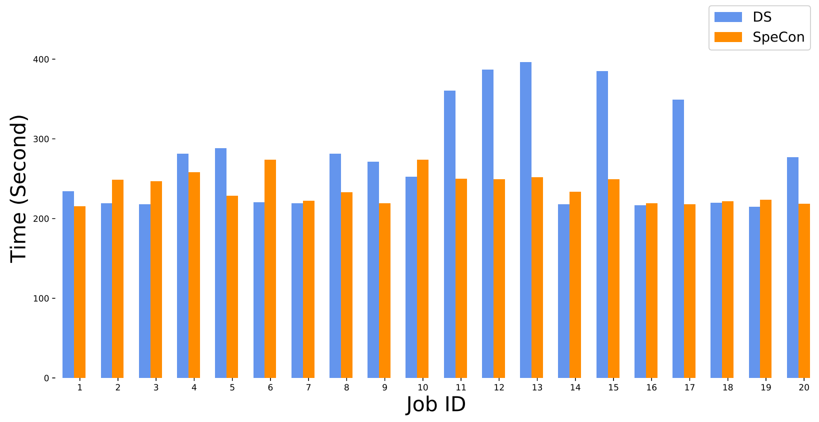


Fig. 6.16: Fixed Schedule with 20 VAE jobs and 50s interval ($\alpha = 0.01, |t_m - t_{m-1}| = 30$)

As shown in Fig. 6.16, we can see that SpeCon outperforms DS in terms of completion time. The largest gain is found on Job-17, which reduces from 348.7s to 217.6s, which is a 37.6% reduction. There are 11 jobs out of 20 that get a reduction in the completion time. The average reduction in those 11 jobs is 83.6s (26.7%). For those 9 jobs that fail to obtain improvement, the average increase is 18.3s (8.1%). Overall, the average completion time of SpeCon reduces from 275.2s to 234.7s (14.7%). With speculative scheduling on slow-growing containers, SpeCon can migrate them to other workers and release resources for fast-growing jobs, which leads to reduction in completion time. As for the makespan of the system, SpeCon reduces the value from 1260s to 1190s.

6.2.3 Random Schedule

Single model

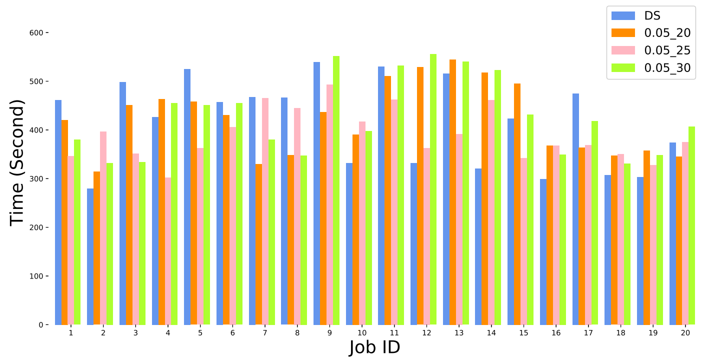


Fig. 6.17: Random schedule [0, 300] with 20 VAE jobs, $\alpha = 0.05$ and varied $|t_m - t_{m-1}|$

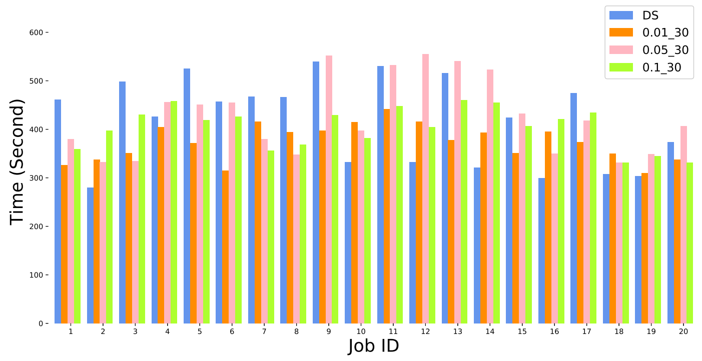


Fig. 6.18: Random schedule [0, 300] with 20 VAE jobs, $|t_m - t_{m-1}| = 30$ and varied α

There are two key parameters used by SpeCon, α that determines the category of each job and

$|t_m - t_{m-1}|$ that decides when to start categorization. Fig. 6.17 and Fig. 6.18 illustrate the performance of SpeCon under various parameter combinations. In these experiments, we randomly submit VAE jobs (single model) within the interval [0, 300] to the cluster.

Fig. 6.17 present results when $\alpha = 0.05$ and $|t_m - t_{m-1}| = 20, 25, 30$. The completion time varies under different parameters. For example, Job-1's completion time is 461.2s, 420.2s, 345.8s, 379.9s for DS, 0.05-20, 0.05-25 and 0.05-30, respectively. SpeCon wins with all testing parameters for Job-1, but degrees of improvements are different. The average completion time of SpeCon improves by 1.1%, 5.2%, and 8.0%.

A similar trend can be found on Fig. 6.18, which plots experiments with $|t_m - t_{m-1}| = 30$ and $\alpha = 0.01, 0.05, 0.1$. The average gain of completion time is 7.2%, 3.2% and 9.5% for $\alpha = 0.01, 0.05, 0.1$, respectively.

Multiple models

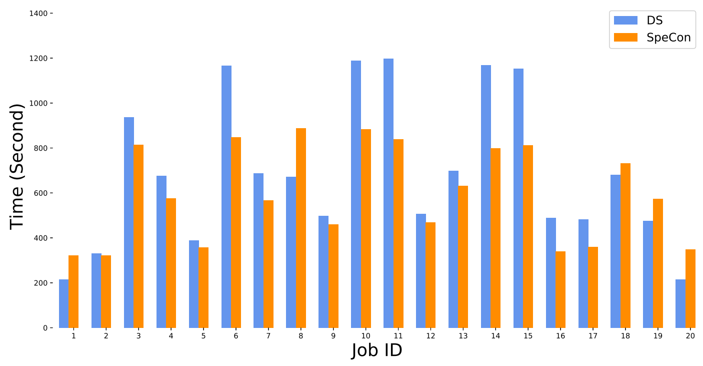


Fig. 6.19: Random schedule [0, 300] with 20 random jobs

Remarks: Comparing to the fixed schedule, degrees of improvements achieved by SpeCon reduced. The reason lies in the fact that random submissions create challenges categorizing the running containers since there is a delay when we monitoring the evaluation function and the single model generates difficulties to select the most desirable worker since the same training pattern has a higher probability leads to the same score for workers. In the rest of the evaluation, the experiments utilize $|t_m - t_{m-1}| = 30, \alpha = 0.01$.

The previous experiments contain only a single model (VAE). Next, we conduct experiments that employ a random selection of both models and platforms in Table 6.3. Fig. 6.19 plots the same cluster with 20 randomly selected jobs submitted to the system within [0, 300].

Overall, there are 15 out of 20 jobs (75%) record a reduced completion time. The largest gain is obtained on Job-14, whose completion time reduced from 1167.8s to 799.2s (31.6%). The average improvement of those 15 jobs is 165.7s. For the other 5 jobs, the average increase is 110.8s. The increase is caused by migrating slow-growing containers to the same worker lead to more resource competition. SpeCon sacrifices a small portion of jobs to improve others significantly. With a system-wide consideration, SpeCon achieves a 13.6% reduction on the average completion time. In terms of makespan in the cluster, SpeCon achieves a 24.7% reduction from 1421s to 1070s.

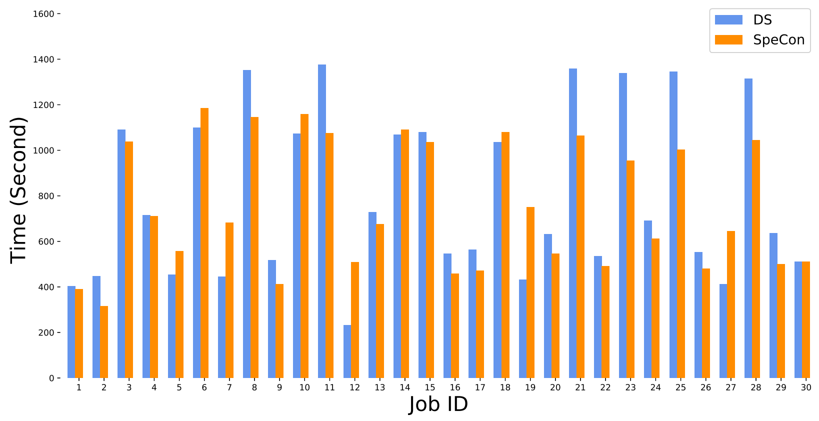
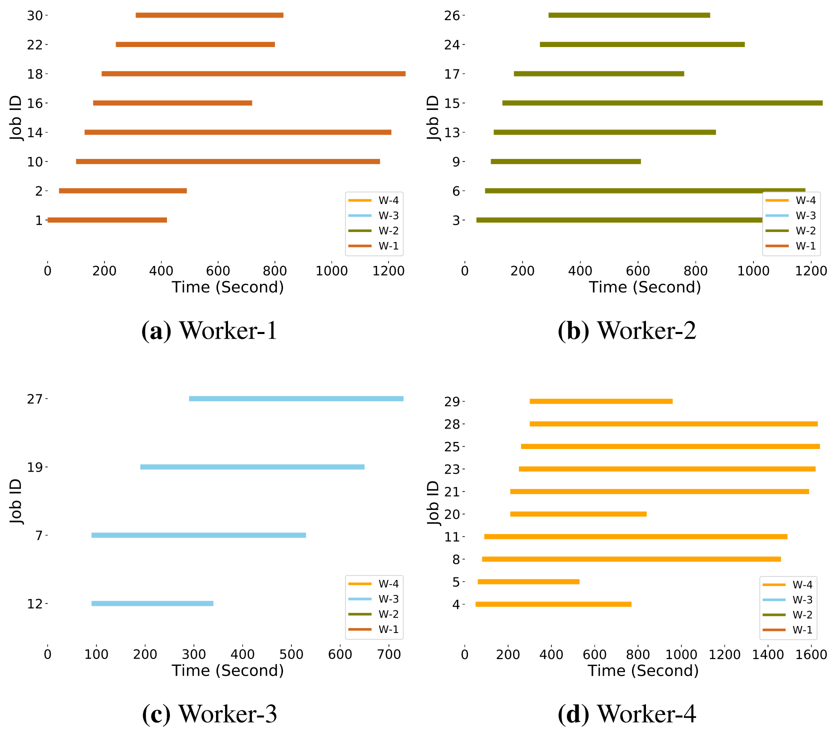


Fig. 6.20: Random schedule [0, 300] with 30 random jobs

Then, we evaluate SpeCon with an increased number of jobs. In this experiment, we launch 30 random jobs within the interval of [0, 300]. Comparing to the previous tests, we increase the density of the job submission by 50%, which creates more challenges to the scheduler. Fig. 6.20 shows the completion time of these jobs. Comparing with DS, 20 out of 30 jobs (66.7%) finish training faster with SpeCon. Job-2 gains the largest, 29.6%, a reduction from 447.4s to 314.9s. Overall, the average completion time improves from 799.5s to 752.6s, 5.9% and makespan reduces from 1640s to 1370s, 16.4%.



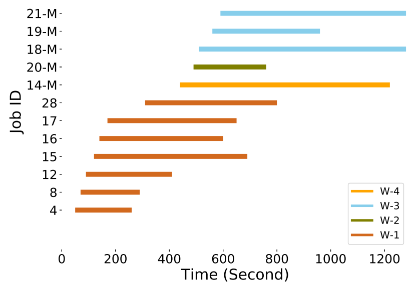
(a) Worker-1 (b) Worker-2

(c) Worker-3 (d) Worker-4

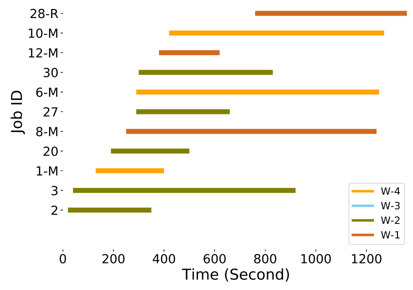
Fig. 6.21: Workload distribution for 30-job experiment with DS

Next, we dig into the details of the experiment. Fig. 6.21 and Fig. 6.22 illustrate the container distribution overtime on each of the workers. In Fig. 6.21, we can clearly see that none of the containers get migrated. With the default scheduler (DS), the migration only happens when the hosting worker is out of resource or other unhealthy conditions. It fails to react based on the job's progress. Moreover, when selecting the host for an incoming container, the default scheduler purely based on the current resource usage (not only CPU but memory, network), which results in an unbalanced distribution on Worker-2, and DS fails to rebalance the workload due to lacking of migration mechanism.

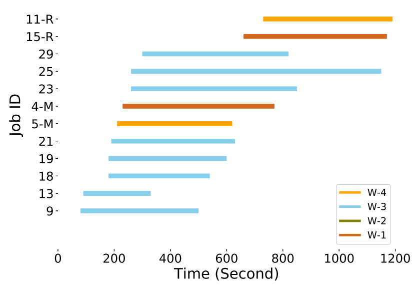
With SpeCon, however, the containers can be migrated according to their training progresses and system-wide workload distribution. In Fig. 6.22, we mark the job that is migrated due to its slow-growing progress as "Job-ID-M" and "Job-ID-R" indicates that this job is migrated as an reaction of system-wide rebalancing. For example, Job-1 was originally running on Worker-4 and it was added to CC category at 155.9s by Algorithm 3. Then, Job-1 was pasted to Algorithm 4. At that moment,



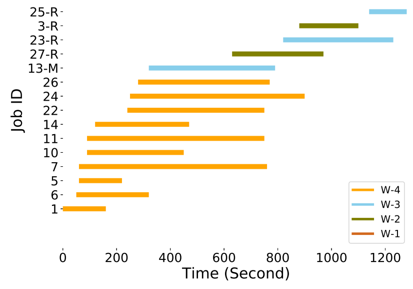
(a) Worker-1



(b) Worker-2



(c) Worker-3



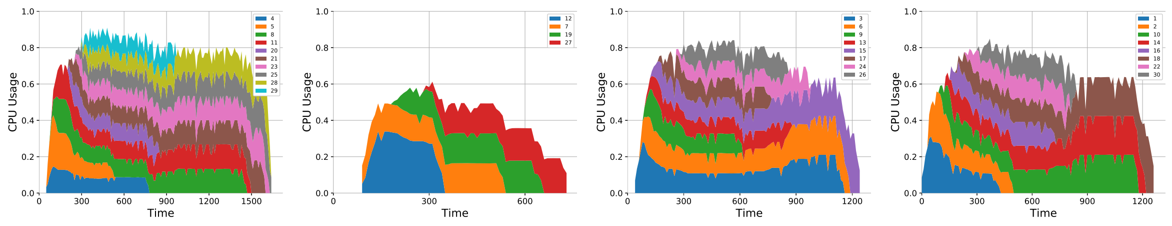
(d) Worker-4

Fig. 6.22: Workload distribution for 30-job experiment with SpeCon

there are 5 other jobs running on Worker-4, and 3, 2, 2 active jobs on Worker-1, Worker-2, Worker-3, respectively. Considering the job's categorization, the weighted scores are 6, 3.5, 4 and 11 for each workers. Therefore, SpeCon selected Worker-2 as a new host for it.

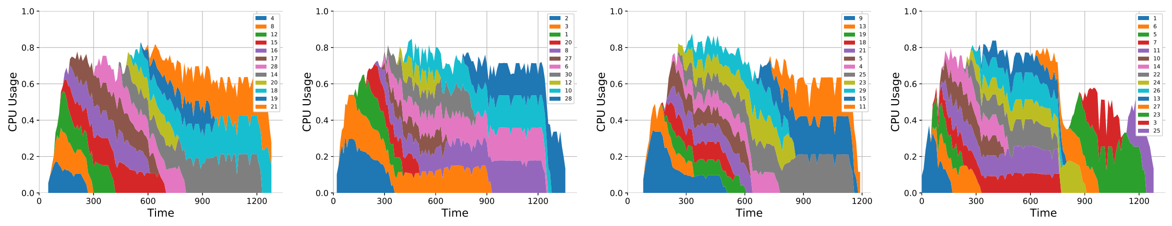
Furthermore, when all the jobs are converged and in CC category, Algorithm 5 start monitoring the cluster. For example, The last job on Worker-4 is finished at 904.5s without considering the rebalancing mechanism. After that, Job-23, 25 from Worker-3 and Job-27, 3 from Worker-2 were reallocated it to increase the system-wide resource utilization and ease the resource competition on Worker-2 and 3. Together, SpeCon results in 66.7% of the jobs (20/30) get a reduced completion time.

Fig. 6.23 and Fig. 6.24 plot the comparison of CPU usage between DS and SpeCon. The resource usage tops at around 80% since Kubernetes reserves part of them for the system itself. Obviously, SpeCon uniformly distributes the workloads across the whole duration. On the contrary, Worker-1 and Worker-3 in DS are always running with high workloads.



(a) Worker-1 **(b)** Worker-2 **(c)** Worker-3 **(d)** Worker-4

Fig. 6.23: CPU usage for 30-job experiment with DS



(a) Worker-1 **(b)** Worker-2 **(c)** Worker-3 **(d)** Worker-4

Fig. 6.24: CPU usage for 30-job experiment with SpeCpn

6.2.4 Larger cluster

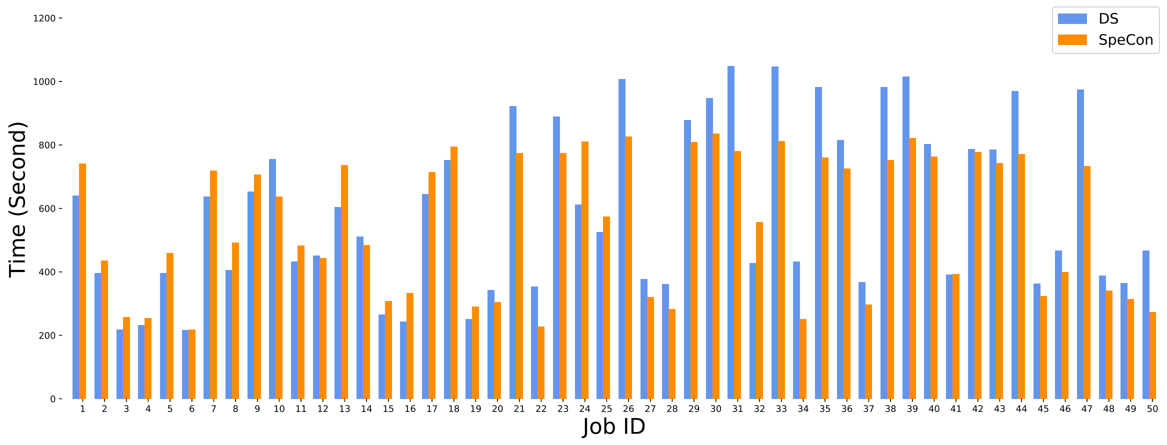


Fig. 6.25: Random schedule [0, 1200s] with 50 random jobs

Finally, we conduct an experiment with a larger cluster, 1 manager and 8 workers. The more workers give more options for SpeCon, which leads to challenges when reallocating containers. Fig. 6.25 present the experiment with 50 random jobs running in the larger cluster. Overall, SpeCon records 30 out of 50 jobs (60.0%) are completing faster than DS. The largest gain is found on Job-50, which reduces from 467.5s to 273.4s which is a 41.5% reduction. The average completion time reduces 7.2% from 596.3s to 553.0s. Additionally, the makespan value reduces from 2070s to 1840s, 11.1%.

CONCLUSION

In this thesis, we try to optimize resource management for containerized deep learning applications through two subproblems: (1) local resource optimization: focus on computing resource configuration for a single machine (or single node); (2) global resource optimization: focus on cluster-wide resource configuration rather than a certain machine's. To address these two subproblems respectively, we introduce two container scheduling system, FlowCon and SpeCon, which aim to facilitate dynamic resource allocation for containerized deep learning training applications at runtime. We have presented the detailed system design of two systems, and conducted extensive experiments with different deep learning models on two frameworks, Pytorch and Tensorflow, in a cloud computing environment. Our experimental results have proven the effectiveness of both FlowCon and SpeCon. Specifically, compared to a current system, FlowCon has achieved significant performance improvement in the presence of various deep learning workloads, by up to 42.06% reduction in completion time for individual jobs without sacrificing the overall system makespan. SpeCon reduces the completion time of an individual job up to 41.5 and achieves the improvement on makespan by up to 24.7% in the presence of different deep learning workloads.

Bibliography

[1] K. D. Foote, *A brief history of deep learning*, 2017. [Online]. Available: `https://www.dataversity.net/brief-history-deep-learning/`.

[2] *Imagenet*. [Online]. Available: `http://www.image-net.org/`.

[3] *Open images dataset*. [Online]. Available: `https://storage.googleapis.com/openimages/web/index.html`.

[4] *Cifar-10*. [Online]. Available: `https://www.cs.toronto.edu/~kriz/cifar.html`.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[8] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[9] Y. You, Z. Zhang, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Imagenet training in 24 minutes," *arXiv preprint arXiv:1709.05011*, 2017.

[10] *Kubernetes*. [Online]. Available: `https://kubernetes.io/`.

[11] *Docker*. [Online]. Available: `https://www.docker.com/`.

[12] W. Zheng, M. Tynes, H. Gorelick, Y. Mao, L. Cheng, and Y. Hou, "Flowcon: Elastic flow configuration for containerized deep learning applications," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[13] Y. Mao, J. Wang, and B. Sheng, "Mobile message board: Location-based message dissemination in wireless ad-hoc networks," in *2016 international conference on computing, networking and communications (ICNC)*, IEEE, 2016, pp. 1–5.

[14] Y. Mao, J. Wang, B. Sheng, and F. Wu, "Building smartphone ad-hoc networks with long-range radios," in *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2015, pp. 1–8.

[15] Y. Mao, J. Wang, J. P. Cohen, and B. Sheng, "Pasa: Passive broadcast for smartphone ad-hoc networks," in *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2014, pp. 1–8.

[16] H. H. Harvey, Y. Mao, Y. Hou, and B. Sheng, "Edos: Edge assisted offloading system for mobile devices," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2017, pp. 1–9.

[17] A. Acharya, Y. Hou, Y. Mao, M. Xian, and J. Yuan, "Workload-aware task placement in edge-assisted human re-identification," in *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, IEEE, 2019, pp. 1–9.

[18] A. Acharya, Y. Hou, Y. Mao, and J. Yuan, "Edge-assisted image processing with joint optimization of responding and placement strategy," in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, IEEE, 2019, pp. 1241–1248.

[19] X. Chen, L. Cheng, C. Liu, Q. Liu, J. Liu, Y. Mao, and J. Murphy, “A woa-based optimization approach for task scheduling in cloud computing systems,” *IEEE Systems Journal*, 2020.

[20] Y. Mao, V. Green, J. Wang, H. Xiong, and Z. Guo, “Dress: Dynamic resource-reservation scheme for congested data-intensive computing platforms,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 694–701.

[21] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, “Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster,” in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2017, pp. 1–8.

[22] Y. Mao, Y. Fu, W. Zheng, L. Cheng, Q. Liu, and D. Tao, *Speculative container scheduling for deep learning applications in a kubernetes cluster*, 2020. arXiv: `2010.11307 [cs.DC]`.

[23] Y. Mao, Y. Fu, S. Gu, S. Vhaduri, L. Cheng, and Q. Liu, *Resource management schemes for cloud-native platforms with computing containers of docker and kubernetes*, 2020. arXiv: `2010.10350 [cs.DC]`.

[24] C.-C. Yang and G. Cong, “Accelerating data loading in deep neural network training,” *arXiv preprint arXiv:1910.01196*, 2019.

[25] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, *et al.*, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 620–629.

[26] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, “Deepx: A software accelerator for low-power deep learning inference on mobile devices,” in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, IEEE, 2016, pp. 1–12.

[27] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, “Large scale distributed deep networks,” in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[28] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[29] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *arXiv preprint arXiv:1508.01991*, 2015.

[30] X. Ma and E. Hovy, "End-to-end sequence labeling via bi-directional lstm-cnns-crf," *arXiv preprint arXiv:1603.01354*, 2016.

[31] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, and B. Xu, "Attention-based bidirectional long short-term memory networks for relation classification," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 207–212. DOI: 10 . 18653 / v1 / P16 - 2034. [Online]. Available: https://www.aclweb.org/anthology/P16-2034.

[32] Y. Tanaka and Y. Kageyama, "Imagenet/resnet-50 training in 224 seconds,"

[33] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi, "Morphnet: Fast & simple resource-constrained structure learning of deep networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 1586–1595.

[34] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in neural information processing systems*, 2014, pp. 1269–1277.

[35] F. Mamalet and C. Garcia, "Simplifying convnets for fast learning," in *International Conference on Artificial Neural Networks*, Springer, 2012, pp. 58–65.

[36] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[37] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 116–131.

[38] R. K. Sharma, P. Kamal, and S. P. Singh, “A latency reduction mechanism for virtual machine resource allocation in delay sensitive cloud service,” in *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, 2015, pp. 371–375.

[39] B. Zhang, Y. Al Dhuraibi, R. Rouvoy, F. Paraiso, and L. Seinturier, “Cloudgc: Recycling idle virtual machines in the cloud,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2017, pp. 105–115.

[40] J.-W. Sung, S.-J. Han, and J.-w. Kim, “Virtual machine pre-provisioning for computation offloading service in edge cloud,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, 2019, pp. 490–492.

[41] P. Kaur and A. Rani, “Virtual machine migration in cloud computing,” *International Journal of Grid Distribution Computing*, vol. 8, no. 5, pp. 337–342, 2015.

[42] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, “A comparative study of containers and virtual machines in big data environment,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 178–185.

[43] Y. Hu, H. Zhou, C. de Laat, and Z. Zhao, “Ecsched: Efficient container scheduling on heterogeneous clusters,” in *European Conference on Parallel Processing*, Springer, 2018, pp. 365–377.

[44] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, “A new container scheduling algorithm based on multi-objective optimization,” *Soft Computing*, vol. 22, no. 23, pp. 7741–7752, 2018.

[45] F. Jiang, K. Ferriter, and C. Castillo, “Pivot: Cost-aware scheduling of data-intensive applications in a cloud-agnostic system,”

[46] W. Zheng, Y. Song, Z. Guo, Y. Cui, S. Gu, Y. Mao, and L. Cheng, “Target-based resource allocation for deep learning applications in a multi-tenancy system,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2019, pp. 1–7.

[47] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, *et al.*, “Gandiva: Introspective cluster scheduling for deep learning,” in *13th*

{*USENIX*} *Symposium on Operating Systems Design and Implementation* ({*OSDI*} *18)*, 2018, pp. 595–610.

[48] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *2019 IEEE International Conference on Big Data (Big Data)*, IEEE, 2019, pp. 278–287.

[49] *Variational auto-encoders*, `http://kvfrans.com/variational-autoencoders-explained/`.

[50] *Bidirectional recurrent neural networks*, `https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks`.

[51] *Gated recurrent unit*, `https://en.wikipedia.org/wiki/Gated_recurrent_unit`.

[52] *Tensorflow*, `https://www.tensorflow.org/`.

[53] *Pytorch*, `https://pytorch.org/`.

[54] *Persistent volume*, `https://kubernetes.io/docs/concepts/storage/persistent-volumes/`.

[55] *Nsf cloudlab*, `https://www.cloudlab.us/`.

[56] *Mnist*, http://yann.lecun.com/exdb/mnist/.

[57] S. Zheng and etc., "Conditional random fields as recurrent neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1529–1537.

[58] M. Berglund and etc., "Bidirectional recurrent neural networks as generative models," in *Advances in Neural Information Processing Systems*, 2015, pp. 856–864.