2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application

2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys) | 978-1-6654-9457-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/HPCC-DSS-SMARTCITY-DEPENDSYS53884.2021.00299

Migrating Deep Learning Data and Applications among Kubernetes Edge Nodes

Suchanat Mangkhangcharoen
Department of Computer Science
Thammasat University
Pathum Thani, Thailand
suchanat.man.mind@gmail.com

Jason Haga
National Institute of Advanced Industrial Science and Technology
Tsukuba, Japan
jh.haga@aist.go.jp

Prapaporn Rattanatamrong
Department of Computer Science
Thammasat University
Pathum Thani, Thailand
rattanat@gmail.com

***Abstract*—Many current IoT applications deployed at the edge use deep learning (DL) in their real-time processing and analytics. Not only inference but also training is moving to edge devices. DL application and dataset migration among these devices are mandatory for scenarios like node failure, user mobility or when nodes need to collaborate (e.g., distributed training). Container technologies and Kubernetes (K8s) are being increasingly adopted to manage infrastructure at the edge. Unfortunately, there is no built-in mechanism in K8s to support migration of stateful containers between its cluster nodes. The K8s cluster's master node generally launches a new fresh container in another node to replace the failed one. While there is an existing mechanism for migrating a Pod between K8s nodes, there is no past work investigating the migration of DL datasets and containerized DL applications among K8s cluster nodes. In this paper, we present our 1) comprehensive study on the effectiveness and limitations of existing checkpointing mechanisms for containerized DL applications and 2) our comparative performance study of several approaches in migrating DL datasets and applications in a K8s cluster. Our results show that migrating states of DL applications and restoring them from their previous states enables faster recovery (reducing training time by 10 to 73 percent) than re-running these models from the beginning regardless of the percentage of epochs that have completed. Additionally, our experimental results show that transferring a dataset between K8s workers using the K8s persistent volume with kubectl cp is generally suitable and efficient. However, when network latency is high, using our customized middleware with a feedback controller to migrate data in parallel can speed up total migration time compared to the K8s's persistent volume approach alone.**

Keywords*—Edge computing, Deep learning application migration, Deep learning data migration, Kubernetes, Fault tolerance

I. Introduction

Internet of things (IoT) has many applications in various fields [1][2][3]. Sensors collect real-time data and edge computers can process the collected data and respond to users faster than servers in the cloud since they are closer to users. Recently, many IoT applications use deep learning (DL) to provide intelligent data processing. Presently, people use edge computing to compute IoT data in addition to the cloud. Edge devices are located closer to users or data sources than cloud servers, so they generally respond to users faster than the cloud servers. Edge computing can be useful in tight real-time applications, such as autonomous vehicles and smart cities, that need to respond extremely fast in real-time and perform AI [4][5][6]. Not only is inference currently moving to the edge, but there is increasing interest to train DL models at the edge [7][8]. In the case of edge node failure, intermittent long-distance network connectivity problem, or malicious harming of edge environments, datasets and applications must also be preserved from being corrupted and allow to continue their operation by replication or migration from an affected node to another node. User mobility and node collaboration (e.g., in distributed training) increase the need for DL application and dataset migration approaches [9][10]. When migrating a DL application, one needs to consider migrating both the model algorithm and the model in-memory state. Also, datasets of DL applications are usually large in size. These pose the main challenges when migration must be done in such resource limited environment like among edge servers.

Container-based virtualization has become the apparent choice to create and execute distributed IoT applications because they are small and provide flexibility [11][12]. Containerizing DL applications can reduce time to deploy them at the edge since containers allow us to reproduce our application environment with required machine learning frameworks or libraries (e.g., Python, TensorFlow [13]) successfully and consistently. This avoids challenges of the deployed devices that could be running different operating systems, kernel versions, GPUs, drivers, and runtimes than our development devices. To manage multiple containers, Google developed an open-source container orchestration, called Kubernetes (K8s) [16]. K8s enables the deployment of high availability containerized applications that can be scaled and managed during runtime with ease [14][15]. A master node in K8s controls multiple worker nodes running application containers inside pods. If one container fails, the master node generally deploys a fresh new container to replace the failed one. Being able to migrate DL applications and resuming their ongoing work is crucial, especially with real-time constraints in extensive DL model training or when collaborative processing among multiple nodes is necessary. Unfortunately, Kubernetes does not have any built-in mechanism to support migration of stateful containers between worker nodes in its clusters, so restoring these containers' processes from their previous states is not possible and must start only from their initial states after migration.

While there are several works studying container migration, there is no past work investigating the migration of DL datasets and stateful DL applications in edge environments where high latency connections exist. This paper presents the following contributions:

- We examined the applicability of using two checkpointing mechanisms in saving training states of DL applications and resuming their executions from the saved states.

978-1-6654-9457-1/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00299

- We conducted a comparative performance study of various methods in migrating DL datasets and applications between a K8s cluster's nodes in an emulated edge environment.

The rest of this paper is organized as follows. Section II provides background and related work. In Section III, we present our study of applying checkpoint and restore of DL application states. Section IV demonstrates our considered Kubernetes edge setup for migrating DL datasets and applications, and then explains how to migrate DL applications in K8s. Section IV also presents several existing techniques that can be used for data migration in K8s. Section V shows the comparative performance study's results for DL dataset and application migration along with our discussion. Finally, Section VI concludes this research and lists possible future work.

II. Background and Related Work

Migrating deep learning applications between different edge servers can be useful for load balancing and accommodating user mobility and it remains one of the open challenges [41]. While there are several works proposed for edge migration for general applications, investigation specifically into deep learning applications has not been considered yet. Our research aims to gain an empirical understanding of whether the program state can be checkpointed and migrated during a DNN execution. The Keras library, one of the most used deep learning frameworks, provides a model checkpointing capability by a callback API [27]. Its ModelCheckpoint callback class allows developers to define where to checkpoint the model weights and load the checkpointed weights in their DL codes. However, this differs from our scenario of interest when we might have little control over the point at which DL applications will be interrupted and must be migrated.

With the popularity of container technologies, creating a stateful snapshot of the migrated container becomes important. This process is called checkpointing. The checkpointed states can be restored later at the destination after migration is completed to reduce the overhead from restarting the application de novo. There are known checkpointing mechanisms to be used with containers, namely Checkpoint Restore in User Space (CRIU) [16] and Distributed MultiThreaded CheckPointing (DMTCP) [17]. CRIU can checkpoint the current state of a container and then restore the container from its previous state in the same host or in another destination host. It is known to work with Docker [18], LXC/LXD [19], Podman [20] and other container technologies for Linux distributions. However, Kubernetes has no support checkpointing pods with CRIU yet, according to [21]. DMTCP proposed by Ansel et al. [23] provides a transparent user-level checkpointing package for distributed applications. Checkpointing and restart is demonstrated for a wide range of over 20 well known cluster and desktop applications (e.g., MATLAB, Python, MPICH2).

Many previous studies focused on container migration and improving the speed of migration. Puliafito et al. [42] studied four migration techniques in a fog environment including pre-copy migration, post-copy migration, hybrid migration and cold migration using CRIU. They concluded that no single container migration technique is the very best under all network and service conditions. Cold migration creates long downtimes, while hybrid migration has a prolonged total migration time. Pre-copy and post-copy migrations represent the best options under different conditions. Ahmed et al. [24] uses DMTCP to checkpoint a docker container to speed up the container's boot phase. However, DMTCP initially runs on x86, x86_64 and mixed (32-bit process in 64-bit Linux) architecture and an experimental port to 64-bit ARM (armv8) was later added as of DMTCP-2.4.0 and there still seems to be some issues to be solved [25]. The MigratingPod library proposed by Schrettenbrunner in [22] implements a mechanism for migrating pods in a K8s cluster using CRIU. The last two works considered only the migration of general web applications and used a more powerful environment than what is typically used in edge computing in their evaluations.

Several works also explore mechanisms to migrate data between nodes. Thongthavorn et al. [26] proposed a middleware framework for migrating multiple containers from one data center to another data center using a feedback controller to dynamically adjust parallel TCP connections. Their results show that this framework can migrate containers between data centers faster than the sequential approach. Junior et al. [43] exploited the OverlayFS file system to transparently snapshot the container's volume contents and transfer them prior to the actual container migration to reduce the container's downtime during migration.

III. Checkpoint and Restore of DL Applications

In this section, we focus on an applicability study of the existing checkpointing mechanisms, namely CRIU and DMTCP to checkpoint training states of DL applications. We examined three DNN models including (1) a sequence-to-sequence RNN model for time series prediction using an encoder-decoder architecture implemented with Python and Tensorflow/Keras [28], (2) a Transformer Network (TF) for trajectory forecasting implemented with Python and PyTorch [29] and (3) the Darknet You Only Look Once (YOLO) v3 model for real-time object detection written in C/C++ and CUDA [30].

For DMTCP, we used an Amazon EC2 t3.medium instance (2 vCPU, Intel x86_64 processor at 3.5GHz and 4GB Memory) with Ubuntu 18.04 OS to run each DNN model listed above and discovered that DMTCP cannot successfully checkpoint DL applications implemented with modern frameworks including TensorFlow, Keras, and PyTorch. It failed to checkpoint both the RNN model and the TF model when connecting to a newly started coordinator. Furthermore, the checkpointed files that were generated after the error occurred could not be used to restore the models' states. The YOLO model is the only one that DMTCP could checkpoint its state and successfully restore it later without any problems. While implementation of DNN models using pure C/C++ is possible, many of today's DL models rely on modern, widely used AI frameworks. Hence, we conclude that DMTCP does not seem to be practical and currently is not suitable for DL checkpointing.

CRIU differs from DMTCP in that CRIU was designed specifically for checkpointing containers rather than regular processes. In this study, we containerized the above three DNN models with Podman, Docker, and containerd [36] running inside the t3.medium instance. Docker uses the containerd daemon to manage container activity on a machine, while Podman uses a daemon-less approach with a technology named conmon [31]. The CRIU checkpointing API for Docker and Podman are provided by [33] and [34],

respectively. CRIU can checkpoint all three models without an error in a Podman container, however, CRIU cannot successfully restore YOLO and gives the error message "Error: OCI runtime error: CRIU restoring failed". There was a period in our study when Docker checkpointing with CRIU did not work properly in Ubuntu due to a kernel issue [32]. After the issue was fixed, the same results as Podman were obtained for Docker. All models running in containerd can be checkpointed, but only RNN and TF can be restored successfully. These results are summarized in Table I and it is obvious that the applicability of CRIU to DL checkpointing depends on the combination of software frameworks and container technologies. From these results, we decided to use the MigratingPod library, which uses CRIU and containerd in our experiment for migration of DL applications in a K8s cluster.

TABLE I. RESULTS OF CHECKPOINTING (C) AND RESTORE (R) DNN MODELS USING THE CRIU APPROACH

DNN Models	Container Technologies					
	Podman		Docker		containerd	
	C	R	C	R	C	R
RNN [28]	✓	✓	✓	✓	✓	✓
TF [29]	✓	✓	✓	✓	✓	-
YOLO [30]	✓	-	✓	-	✓	✓

Like DMTCP, it would still be challenging to get CRIU to work with non-x86 architectures (according to [16]). Since both checkpointing mechanisms considered here do not officially support the 64-bit ARM architecture, and this is the main cause of their failures, we believe there is still a research gap and opportunity for checkpointing with the 64-bit ARM architecture; many edge hardware use this architecture, e.g., Raspberry Pi, NVIDIA Jetson Nano, TX1. In some edge hardware with limited computing resources, like the Raspberry Pi, it is still more practical to deploy DL applications for inference only and not for training from our experiment's result.

IV. MIGRATING DL DATASETS AND APPLICATIONS IN A KUBERNETES EDGE CLUSTER

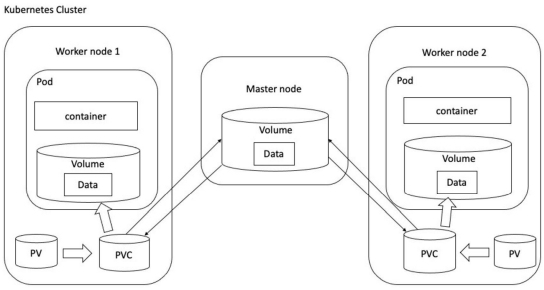

Fig. 1. A K8s cluster setup for DL dataset and application migration

In this research, we are interested in a scenario (as shown in Fig.1) where the entire K8s cluster is deployed at the edge or a combination of edge and cloud environments where a master node is in the cloud and worker nodes are at the edge. Inside each worker node is a pod with a container running a DL application inside. Each pod uses a PersistentVolumeClaim (PVC) for accessing a persistent storage, called PersistentVolume (PV). A DL dataset is stored inside this hostpath PV. When a node fails, its pod with the DL application container and the dataset residing in the PV are migrated to another node in the same cluster. In some cases, only the DL application or the dataset might need to be migrated.

A. Migration of a DL Application using MigratingPod

The MigratingPod library [22] requires that the master node has a Network File System (NFS) server, and each worker node must be an NFS client as shown in Fig 2. In addition, containerd must be installed in the K8s cluster [35], which is different from the typical K8s setup using Docker containers. Containerd is a container runtime that can be used as a daemon to manage a container lifecycle for Linux and Windows [36].

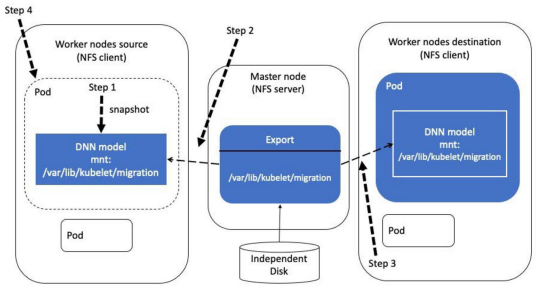

Fig. 2. Steps in migrating DL application among K8s workers.

The command "kubectl migrate <pod name> <destination node name>" starts migration of the specified pod to the destination node. As illustrated in Fig. 2, steps of the DL application migration by MigratingPod are as follows:

1) MigratingPod uses CRIU to snapshot the state of the container on the pod to be migrated.

2) A dump file from the CRIU snapshot is then exported from the worker node and it is available to the master node via NFS mount.

3) A new pod is subsequently cloned in the destination node, and the dump file can be accessed at the destination node via NFS mount. The new pod restores its state from this dump file and is now in the running state.

4) The old pod is removed from the source node, completing the migration.

B. Approaches for Migrating a DL dataset

There are several approaches that we can utilize to move a dataset from one node to another in a K8s cluster. The seemingly most straightforward approach is to use the Linux file transfer command 'scp'. However, scp transfers files between nodes directly, so the orchestrator node will not be aware of what is going on between nodes that it manages. The second approach is more suitable for cluster orchestration since it uses the K8s' 'kubectl cp' command to copy the content inside the Pod's PV to the other node. Migrating data between workers using the latter approach however must be done via the master node only, since worker nodes generally cannot communicate with each other directly with K8s.

For the third approach, we adopted the middleware framework previously proposed in [26] to speed up migration

based on the second approach. The middleware was originally created for migrating multiple containers from one data center to another data center using dynamic numbers of parallel TCP connections and its prototype implements a migrator for LXC/LXD containers that users need to specify the number of containers to be migrated manually. The middleware uses a feedback controller to set the value of the parallel window indicating how many TCP parallel connections should be used to transfer files in each round based on the current network condition. We implemented a new migrator that works with the K8s 'kubectl cp' command for this middleware and enabled it to automatically retrieve the list of all file names and the total number of files to be migrated. The feedback controller also retrieves network throughput from the K8s kubectl cp's output. In addition, we added a function to handle lost packets that was not considered in the original prototype. If any file failed to reach its destination, it is added back to a resend list and queued for the middleware to send them to the destination again.

V. A Comparative Performance Study

To compare the performance of migration approaches for DL datasets and applications, we used an emulated edge environment by setting up a Kubernetes cluster (illustrated in Fig. 3) with the following node configurations:

- A master node is an Amazon EC2 t3.medium instance (2 vCPU, 4 GB Memory) with the disk size 30 GB. It has installed Ubuntu Server 18.04, Containerd version 19.03.12, and Kubernetes version 19.04. We installed the middleware for data migration and an NFS server in the master node.
- Two worker nodes: each node is an Amazon EC2 t3.large instance (2 vCPU, 8 GB Memory) with the disk size 20 GB. Each node has installed Ubuntu Server 18.04, Containerd version 19.03.12, and Kubernetes version 19.04. Workers also serve as NFS clients and have relevant mount points with the master node. These two workers reside in the same availability zone, but the master node is in a different zone from these two worker nodes.

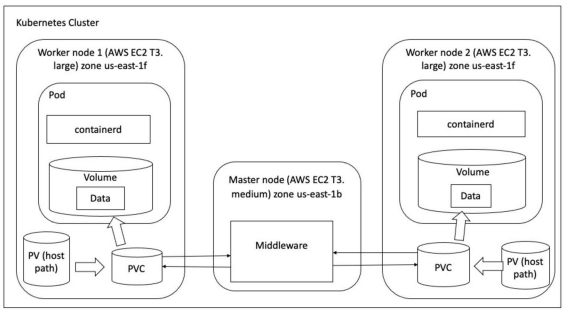

Fig. 3. An emulated K8s edge cluster setup.

Network latency between nodes within the K8s cluster that we considered in our experiments are emulated with the Linux 'tc' command at 10ms, 20ms and 30ms, based on the network latency of edge environments for augmented Reality (AR) and Intelligent transport systems mentioned in the Intel white paper [37]. For the data migration, we added a network latency of 70ms to investigate the behavior of time to migrate and ensure that it is the fastest with increased latency.

A. *Effectiveness and Efficiency of DNN migration in K8s using MigratingPod*

In this experiment, we studied the migration of four different DNN models as shown in Table II: (1) the RNN model from the experiment in Section III, (2) the CNN model for image detection using TFLite [38] (denoted as CNN1), (3) the Convolution Neural Network for optical character recognition [39] (denoted as CNN2) and (4) the machine translation using Encoder-Decoder LSTM [40].

First, we recorded the accuracy/loss values of the models after the training completes both with migration and without migration. The results are shown in the last two columns of Table II. The accuracy/loss values in both cases are the same for all models. This demonstrates that the ongoing state of DNN training can be captured correctly and migrated, and the migration does not affect the model accuracy when restored in the destination host. Hence, the migration in K8s using MigratingPod is considered effective.

TABLE II. Information of Deep Learning Models and Accuracy/Loss Results Without and With Migration.

DNN Models	Model Information			Accuracy/Loss	
	Model size	*Batch size*	*Container Image Size*	*Without migration*	*With migration*
RNN	397KB	512	1.29GB	Loss = 0.0224	Loss = 0.0224
CNN1	2.7MB	32	1.89GB	Accuracy = 0.8257	Accuracy = 0.8257
CNN2	4.7MB	32	1.27GB	Accuracy = 0.7678	Accuracy = 0.7678
LSTM	32MB	64	1.3GB	Loss = 0.4929	Loss = 0.4929

Next, we compared the total training time when (1) migrating the models and resuming their training at the destination, and (2) launching a new container and restarting the models' training at the destination. We denote the total training time for case (1) as Migrate Training Time (MTT) and the total training time of case (2) as New container Training Time (NTT). We considered the cases when failure occurs during the training of each model in the source node and then migrated the model after 25%, 50% and 75% of all epochs were completed. For example, if the training requires a total of ten epochs, the model is migrated at 50% of the training after completing the fifth epoch. We also measured down time when migrating (MDT) and down time when launching a new container (NDT). The down time is defined as the period from when the node failure occurs to when the model resumes its training at the destination. We repeated the experiment three times for each setting. Here, we assume that the container image already exists in the destination node. Fig. 4 shows the average MTT, NTT, MDT and NDT and their standard deviations for all cases of all models.

As expected, migrating a DL model during its training and resuming its operation at the destination yields a shorter total training time when compared to launching a new instance and restarting training from the beginning. The difference between the corresponding MTT and NTT pairs increases as the amount of training epochs to repeat (i.e., the percentage of trained epochs before node failure occurs). As the container image size grows, the down time of the migration also becomes longer because of more bytes to migrate. The CNN1 model has the longest MDT since it has the largest container image of all models. The down time of launching a new container (NDT) does not vary with the container image size

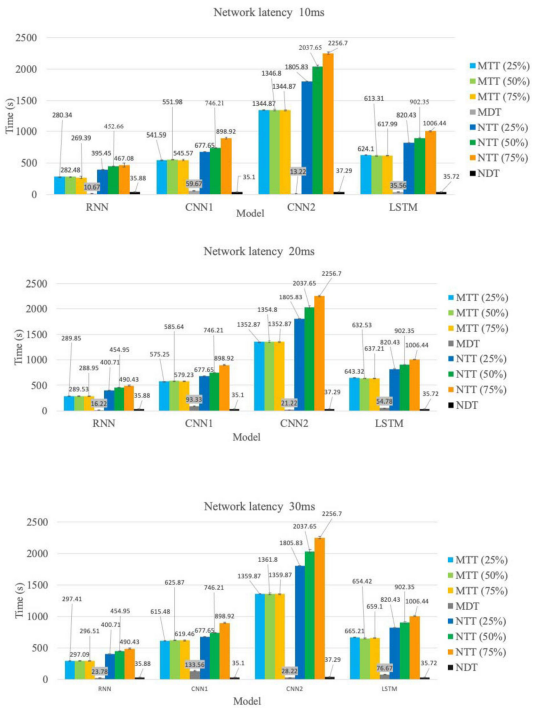

Fig. 4. Averages and standard deviations of total training time when migrating from the source node (MTT) and when launching a new container (NTT) in seconds with network latency of 10ms, 20ms and 30ms.

since we assume that the container image already exists in the destination node. Table III shows the increase of training time when launching a new container instead of migrating from the source node and resuming its operation at the destination node. Ongoing training in this case were lost and the training must be restarted from the beginning. We reported the minimum and maximum values of cases when migrating at 25%, 50% and 75% of all epochs. The high percentage increase confirms the benefit of the migration instead of a new container launch. As the network latency between the source and destination nodes increases, the benefit gained from migration degrades as it takes longer time to migrate. For most models except CNN2, the benefit drops by about 10 percent. The CNN2 model takes the most time to train, so the benefit is still considered significant although the network latency becomes higher. Overall, migrating the models and resuming their operation can avoid at least 10.10 percent and up to 73.38 percent increase in the total training time.

TABLE III. PERCENTAGE INCREASE IN TRAINING TIME IF LAUNCHING A NEW CONTAINER INSTEAD OF MIGRATING FROM THE SOURCE NODE.

DNN Models	10ms		20ms		30ms	
	Min	*Max*	*Min*	*Max*	*Min*	*Max*
RNN	41.06	73.38	38.25	69.73	34.73	65.40
CNN1	25.12	64.77	17.80	55.19	10.10	45.11
CNN2	34.28	67.80	33.48	66.81	32.79	65.95
LSTM	31.46	62.86	27.97	58.5	23.33	52.70

B. DL Dataset Migration in K8s

In this section, we examined four different approaches to handle the case when a dataset needs to be moved to another node. The first approach is to download the dataset from its original Internet storage or repository (denoted as Repo d/l). This is usually a typical approach used in general settings. The second approach is to use the Linux file transfer command 'scp' to directly move the dataset from its original node to the destination node (denoted as scp). For the third approach, the 'kubectl cp' is used together with PV (denoted as kubectl cp). Lastly, we extended the middleware previously proposed by [26] to dynamically control the number of parallel 'kubectl cp' transfers. To study how efficient each of the four mechanisms performs in a K8s cluster, we use three different datasets shown in Table IV. The time to move the whole dataset to the destination node using each approach was recorded. For the latter two approaches, we recorded two separate cases: when the source node is the master node (m-w), and when the source node is a worker node (w-w). The destination node is always a worker node. The experiment was repeated three times for each setting. The averages and standard deviations of total data migration time from these experiments are shown in Fig. 5.

TABLE IV. INFORMATION ABOUT THREE DATASETS TO BE MIGRATED.

Dataset	Data type	Total size	Number of files	Average/Stdev. of file size
Imagenet [44]	Images	2.6 GB	10	376MB / 0B
TCWikipedia [45]	Text	1.1 GB	280	93.59MB / 25.09B
ESC-50 [46]	Audio	600 MB	1000	432KB / 0B

In a low latency network (i.e., 10ms and 20ms latency), using the K8s native mechanism for transferring data between nodes (i.e., kubectl cp) appears the most efficient for the ESC-50 dataset, which consists of a large number of small-size files. Using the kubectl cp approach, K8s creates a Tar archive of the files and copies it over the network. Since Tar can reduce space usage when used on many small files that are smaller than the filesystem's cluster size [47], we observed better performance than scp. For the other datasets whose file size is larger and fewer files, the performance of kubectl cp and scp are comparable. Direct downloading of the two smaller datasets from the repository takes a longer time than the kubectl cp approach since the repository could be far away from the edge node and hence takes longer time to download. The download time in this experiment is still considered quite acceptable for some applications. For the largest dataset, downloading from data repository approach performs much worse than any other approaches due to the same reason about the distance from the edge node and the repository and the increased amount of data to download. Using the middleware does not seem to speed up the K8s data transfer in this setting where the network latency is low.

The time to migrate between workers generally takes twice as much the time to migrate from the master to a worker since workers cannot communicate directly through the K8s API. This is also the reason why the scp approach outperforms the kubectl cp approach for the Imagenet and TCWikipedia datasets because the worker-worker migration can be done without a mediator. However, the performance obtained using the scp approach is not as stable as the kubectl cp approach;

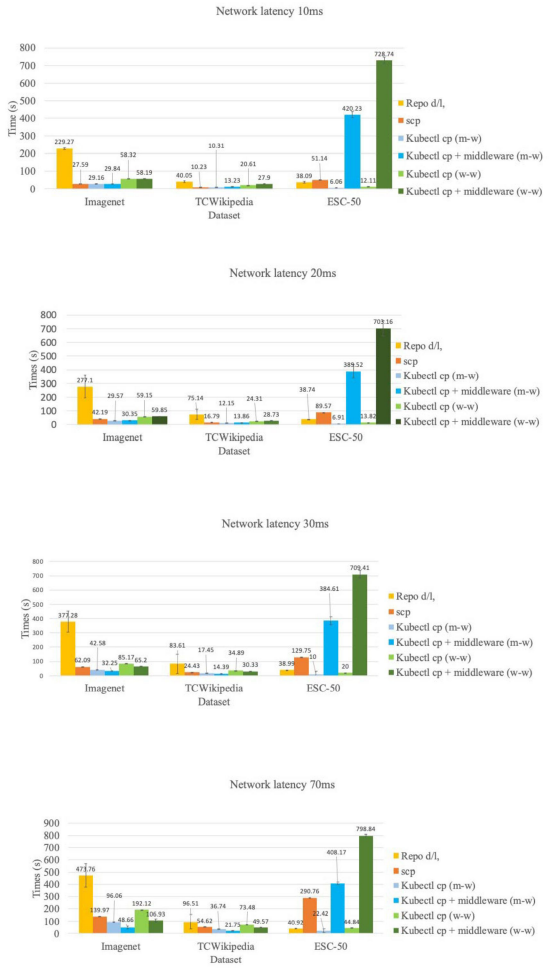

Fig. 5. Averages and standard deviations of total data migration time when migrating from the source node (MTT) and when launching a new container (NTT) in seconds with network latency of 10ms, 20ms, 30ms and 70ms.

we observed that the standard deviation using scp can be rather high.

For the networks with higher latency (i.e., 30ms and 70ms), the results are still mostly similar with the low latency network settings. This scenario is likely when the cluster of edge servers is distributed over a wider geographical area and controlled by a master node that is further away. However, the middleware approach seems to help in speeding up the kubectl cp approach (compare the light blue and dark blue bars and the light green and dark green bars for higher latencies in Fig. 5). The improvement in migration time is especially obvious (up to about 49% decrease) for the 70ms latency setting. This is due to the adaptive parallel migration of files enabled by the middleware's mechanism. We can also see significant degradation in performance of the scp approach when the latency becomes extremely high. One interesting result that we observed is that the middleware approach performs poorly for the ESC-50 dataset although it is the smallest dataset of all the datasets that we used in the experiment. After a careful inspection of the dataset, we found that the ESC-50 dataset has the largest number of rather small files. Since the middleware uses one TCP connection per file the overhead incurred by the TCP connection setup exceeds the data payload explaining the poor performance in this case. In conclusion, the middleware can provide improvement only for large datasets with moderate file size in the high latency network setting. Otherwise, the K8s kubectl cp approach is efficient and appropriate.

VI. Conclusions

In this paper, we present our study in applying the existing checkpointing mechanisms, namely DMTCP and CRIU, to deep learning applications. While CRIU can checkpoint the containerized DL applications in x86_64 computers, its functionality is still highly sensitive to the OS kernel and container technology used. We also examined the effectiveness and efficiency of the MigratingPod mechanism, which uses the CRIU checkpointing, in migrating DL applications during their training process. The result shows that the MigratingPod mechanism can migrate the state of DNN training without affecting the accuracy of the models and the migration can avoid at least 10 or up to 73 percent of training time increase when compared to the typical mechanism in K8s that launches a new container to train from the beginning. For data migration, using the K8s kubectl cp command with PV is the most appropriate and predictable approach for migrating a dataset in a K8s cluster. However, for large datasets and high latency networks, the middleware that can dynamically migrate data files in parallel appears to help reducing the total data migration time by up to 51.74 percent.

Acknowledgment

The authors gratefully acknowledge the assistance of W. Thongthavorn for his advice in modifying the original middleware prototype and J. Schrettenbrunner for his help setting up MigratingPod in our experiment environment.

References

[1] S. Nižetić, et al., Internet of Things (IoT): Opportunities, issues and challenges towards a smart and sustainable future, Journal of Cleaner Production, Volume 274, 2020, 122877, ISSN 0959-6526, https://doi.org/10.1016/j.jclepro.2020.122877.].

[2] C. Zhang, Design and application of fog computing and Internet of Things service platform for smart city, Future Generation Computer Systems, Volume 112, 2020, Pages 630-640, ISSN 0167-739X, https://doi.org/10.1016/j.future.2020.06.016.

[3] A. D. Boursianis, et al., Internet of Things (IoT) and Agricultural Unmanned Aerial Vehicles (UAVs) in smart farming: A comprehensive review, Internet of Things, 2020, 100187, ISSN 2542-6605, https://doi.org/10.1016/j.iot.2020.100187.

[4] J. Zhang and K. B. Letaief, "Mobile Edge Intelligence and Computing for the Internet of Vehicles," in Proceedings of the IEEE, vol. 108, no. 2, pp. 246-261, Feb. 2020, doi: 10.1109/JPROC.2019.2947490.

[5] S. Chakrabarty and D. W. Engels, "Secure Smart Cities Framework Using IoT and AI," 2020 IEEE Global Conference on Artificial Intelligence and Internet of Things (GCAIoT), 2020, pp. 1-6, doi: 10.1109/GCAIoT51063.2020.9345912.

[6] Z. Lv, et al., AI-enabled IoT-Edge Data Analytics for Connected Living. ACM Trans. Internet Technol. 21, 4, Article 104 (November 2021), 20 pages. DOI:https://doi.org/10.1145/3421510.

[7] X. Wang, et al. Edge AI: Convergence of Edge Computing and Artificial Intelligence. Springer Nature, 2020.

[8] F. Foukalas and A. Tziouvaras. "Edge AI for Industrial IoT Applications." IEEE Industrial Electronics Magazine (2021).

[9] Q. Xia, et al., A survey of federated learning for edge computing: Research problems and solutions, High-Confidence Computing, Volume 1, Issue 1, 2021, 100008, ISSN 2667-2952, https://doi.org/10.1016/j.hcc.2021.100008.

[10] Q. Han, et al., "OL4EL: Online Learning for Edge-Cloud Collaborative Learning on Heterogeneous Edges with Resource Constraints," in IEEE Communications Magazine, vol. 58, no. 5, pp. 49-55, May 2020, doi: 10.1109/MCOM.001.1900594.

[11] Alam, M., Rufino, J., Ferreira, J., Ahmed, S.H., Shah, N., Chen, Y.: Orchestration of microservices for IoT using docker and edge computing. IEEE Commun. Mag. 56(9), 118–123 (2018).

[12] Javed, A., Robert, J., Heljanko, K. et al. IoTEF: A Federated Edge-Cloud Architecture for Fault-Tolerant IoT Applications. J Grid Computing 18, 57–80 (2020). https://doi.org/10.1007/s10723-019-09498-8.

[13] "TensorFlow," [Online]. Available: https://www.tensorflow.org/versions. [Accessed 16 09 2021].

[14] Kristiani, E., Yang, CT., Huang, CY. et al. The Implementation of a Cloud-Edge Computing Architecture Using OpenStack and Kubernetes for Air Quality Monitoring Application. Mobile Netw Appl 26, 1070–1092 (2021). https://doi.org/10.1007/s11036-020-01620-5.

[15] C. Dupont, R. Giaffreda and L. Capra, "Edge computing in IoT context: Horizontal and vertical Linux container migration," 2017 Global Internet of Things Summit (GIoTS), 2017, pp. 1-4, doi: 10.1109/GIOTS.2017.8016218.

[16] "CRIU," [Online]. Available: https://criu.org. [Accessed 02 11 2020].

[17] "DMTCP: Distributed MultiThreaded CheckPointing," [Online]. Available: http://dmtcp.sourceforge.net. [Accessed 16 09 2021].

[18] “Docker”, https://www.docker.com/

[19] "Container and virtualization tools," [Online]. Available: https://linuxcontainers.org. [Accessed 16 09 2021].

[20] "Podman," [Online]. Available: https://podman.io. [Accessed 16 09 2021].

[21] “Kubernetes Issues: Pod lifecycle checkpointing,” [Online]. Available: https://github.com/kubernetes/kubernetes/issues/3949

[22] J. Schrettenbrunner, Jakob, Migrating Pods in Kubernetes, Master These, 2020, 10.13140/RG.2.2.31821.97762.

[23] J. Ansel, K. Arya and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," 2009 IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1-12, doi: 10.1109/IPDPS.2009.5161063.

[24] A. Ahmed, et al., "Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart," 2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2020, pp. 55-62, doi: 10.1109/MobileCloud48802.2020.00016.

[25] “DMTCP FAQ: Architectures Supported by DMTCP,” [Online]. Available: http://dmtcp.sourceforge.net/FAQ.html

[26] W. Thongthavorn and P. Rattanatamrong, "Multi-Container Application Migration with Load Balanced and Adaptive Parallel TCP," 2019 International Conference on High Performance Computing & Simulation (HPCS), 2019, pp. 55-62, doi: 10.1109/HPCS48598.2019.9188218.

[27] “Keras API reference / Callbacks API / ModelCheckpoint,” [Online]. Available: https://keras.io/api/callbacks/model_checkpoint/.

[28] L. Tonin, “Keras implementation of a sequence to sequence model for time series prediction using an encoder-decoder architecture,” [Online]. Available: https://github.com/LukeTonin/keras-seq-2-seq-signal-prediction.

[29] Giuliari, Francesco & Hasan, Irtiza & Cristani, Marco & Galasso, Fabio. (2020). Transformer Networks for Trajectory Forecasting, International Conference on Pattern Recognition (ICPR), 2020.

[30] “Darknet: Open Source Neural Networks in C,” [Online]. Available: https://pjreddie.com/darknet/.

[31] S. Abraham, A. K. Paul, R. I. S. Khan and A. R. Butt, "On the Use of Containers in High Performance Computing Environments," 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 284-293, doi: 10.1109/CLOUD49709.2020.00048.

[32] “checkpoint-restore/criu issues: error when checkpointing container: Can’t lookup mount,” [Online]. Available: https://github.com/checkpoint-restore/criu/issues/860.

[33] “Docker checkpoint,” [Online]. Available: https://docs.docker.com/engine/reference/commandline/checkpoint/.

[34] “Podman checkpointing,” [Online]. Available: https://podman.io/getting-started/checkpoint

[35] “Kubernetes: Container Runtime,” [Online]. Available: https://kubernetes.io/docs/setup/production-environment/container-runtimes/.

[36] “Containerd,” [Online]. Available: https://containerd.io/#:~:text=containerd%20is%20available%20as%20a,to%20network%20attachments%20and%20beyond.

[37] Intel, [Online]. Available: https://builders.intel.com/docs/networkbuilders/low-latency-5g-upf-using-priority-based-5g-packet-classification.pdf.

[38] Mjrovai, "TFlite AI At The Edge," Github, 19 08 2020. [Online]. Available: https://github.com/Mjrovai/TFLite_IA_at_the_Edge . [Accessed 14 09 2021].

[39] R. Gandhi, "Build Your Own Convolution Neural Network in 5 mins," towards data science, 19 04 2018. [Online]. Available: https://towardsdatascience.com/build-your-own-convolution-neural-network-in-5-mins-4217c2cf964f. [Accessed 14 09 2021].

[40] prakhargurawa. [Online]. Available: https://github.com/prakhargurawa/Neural-Machine-Translation-Keras-Attention. [Accessed 09 07 2021].

[41] J. Chen and X. Ran, "Deep Learning With Edge Computing: A Review," in Proceedings of the IEEE, vol. 107, no. 8, pp. 1655-1674, Aug. 2019, doi: 10.1109/JPROC.2019.2921977.

[42] C. Puliafito, et al. (2019). Container Migration in the Fog: A Performance Evaluation. Sensors. 19. 1488. 10.3390/s19071488.

[43] D. M. G. P. Paulo Souza Junior, "Stateful Container Migration in Geo-Distributed Environments," in *12th IEEE International Conference on Cloud Com- puting Technology and Science*, Bangkok, Thailand., 2020.

[44] Download ImageNet Data [Online]. Available: https://image-net.org/download-images.php

[45] Tagged and Cleaned Wikipedia (TC Wikipedia) and its Ngram, [Online]. Available: https://nlp.cs.nyu.edu/wikipedia-data/Intel,

[46] ESC-50, [Online]. Available: https://github.com/karolpiczak/ESC-50

[47] How does tar work in Linux? [Online]. Available: https://ostoday.org/linux/how-does-tar-work-in-linux.html