Architecture to Distribute Deep Learning Models on Containers and Virtual Machines for Industry 4.0*

Jean-Sébastien Lerat*†
Jean-Sebastien.Lerat@umons.ac.be
*Science and Technology Department
Haute école en Hainaut
Mons, Belgium

Sidi Ahmed Mahmoudi†
Sidi.Mahmoudi@umons.ac.be
†Computer Science and Management Department
University of Mons
Mons, Belgium

***Abstract*—Deep learning (DL) is increasingly used in industry, especially in industry 4.0. Thanks to DL, it possible to better prevent breakdowns and manufacturing defects. DL models are becoming more and more complex and efficient, requiring significant compute resources and compute time. The use of Graphic Processing Units (GPUs) makes it possible to speed up processing but at a higher cost. An alternative to them is the use of distributed DL (DDL) which differs from Federated Deep Learning in that it focuses on accelerating calculations and does not address data privacy. DLL requires having several computing nodes. This is where cloud computing comes in. Cloud computing allows resources or virtual machines to be allocated on demand, which reduces costs. However, the allocation of GPU resources has a higher cost than CPU resources, which can be problematic for small businesses. This article proposes to exploit the DDL on CPUs via the on-demand allocation of virtual machines in order to reduce costs. In addition, a solution for deploying the software stack necessary for proper operation is proposed. This is achieved using a containerization which is only composed of the software suites needed to run the DDL to minimize the container transfer size and consequently minimize the container deployment time.**

***Index Terms*—cloud computing, high performance computing,distributed deep learning,container, industry 4.0**

I. Introduction

Deep learning (DL) is increasingly used in industry, especially in Industry 4.0. More and more companies are using deep learning (DL) models [1] to prevent breakdowns and manufacturing defects [2]. DL models are becoming more and more complex and efficient, requiring significant compute resources and compute time. With the growth of data and the complexity of new neural networks, higher computing power capabilities are required. As a response, new GPU devices are designed and are able to quickly execute DL tasks [3]. Another approach is distributed DL (DDL) which consists of a multi-node architecture that is exploited to train a DL model or to infer. The aim is to minimize computation time. Unlike DDL, Federated Deep Learning (FDL) focuses on pooling data to train a common, more accurate model. In FDL, entities retain their own data to guarantee privacy, and each entity trains locally its own model. Then each of them sends updates to the common model. FDL thus exploits a form of DDL where the main objective is not to minimize processing time. A report [4] shows that distributed DL (DDL) is used to train DL models even faster. However, industries rarely have GPUs or a physical high-performance computing (HPC) architecture that is expensive. Instead, they often own cloud servers or use cloud providers. In this paper, we propose to train deep learning models on virtual machines with the minimal stack of software inside Docker containers. We also propose to efficiently apply data parallelism in CPUs with the Message Passing Interface (MPI) framework. We show that quick computation can be achieved with cheap virtual machines (VMs) equipped with multiple CPUs. This is a solution for small businesses for which HPC or specific GPU hardware is not affordable.

The remainder of the paper is organized as follows: Section 2 presents the related work, while Section 3 presents our proposed deployment architecture. Experiments and results are respectively presented in the fourth and fifth sections. Finally, the last section is devoted to presenting conclusions and future work.

This work was partially funded by Fédération Wallonie-Bruxelles (Jagang). We would like to thank the company EASI (Belgium) which kindly made resources available to carry out these analyses.

II. Related Work

The related work section is organized into two subsections: technology overview and model deployment and training.

A. Technology Overview

Docker [5] is an open-source tool for developing, shipping, and running applications using containerization. It provides operating system-level virtualization or application-level virtualization, allowing applications to be isolated from their environment. Docker achieves isolation through the use of the GNU/Linux namespaces system. Figure 1 illustrates a host running n Docker containers on a GNU/Linux kernel.

Each aspect of a container operates within a separate namespace, providing limited access only to that namespace. The namespaces include pid, net, ipc, mnt, and ufs. By leveraging namespaces, processes running inside a namespace believe that they have their own isolated instance of the corresponding resource. Additionally, each container is associated with its own control group (cgroup) that defines resource usage constraints for the application. Control groups enable Docker to enforce limits and restrictions on available hardware resources while also facilitating resource sharing. For example, Figure 2

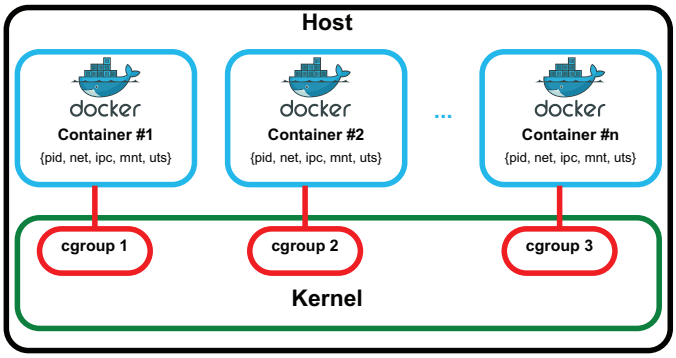

Fig. 1. The schema depicts n containers running on a single host, each with its own namespaces. The blue rectangles represent the Docker containers, the red rectangles represent the control groups of the containers, and the green rectangle represents the GNU/Linux kernel.

illustrates n control groups that jointly share and regulate the utilization of CPU, memory, devices, input/output, and the process tree.

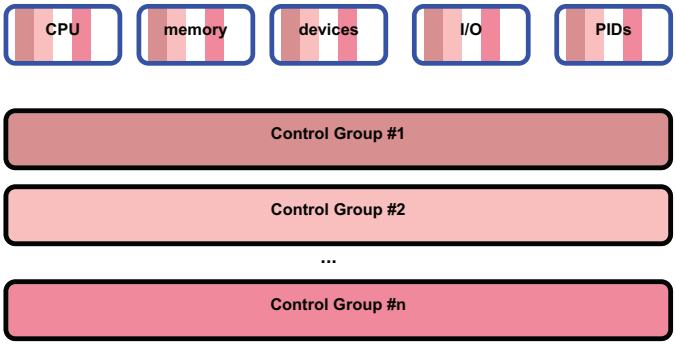

Fig. 2. Schema of the GNU/Linux control group feature. Each distinct color represents a group with its control over each type of resource.

The isolation mechanisms of containers rely on kernel features, which are software components. An alternative approach to containers is virtualization, with hardware-assisted virtualization being the most efficient type since 2005. This form of virtualization leverages hardware capabilities to provide an isolation mechanism based on protection rings, as depicted in Figure 3. The CPU architecture supports specific operation codes (opcodes) that allow the hypervisor to operate in ring -1, a privileged ring with complete control over the hardware. The guest kernel, running in ring 0, retains full control over the virtual hardware. The privilege level decreases with higher ring values, indicating reduced code privileges.

Due to the hardware-based implementation of isolation, virtualization offers enhanced security compared to containers and tends to exhibit faster execution. However, containers facilitate denser deployment since multiple isolated applications can run on a single operating system, whereas virtual machines require a guest operating system for each instance.

To manage and orchestrate containers, there are two software solutions: Docker Swarm mode and Kubernetes. Docker Swarm mode allows you to manage Docker engine clusters lo-

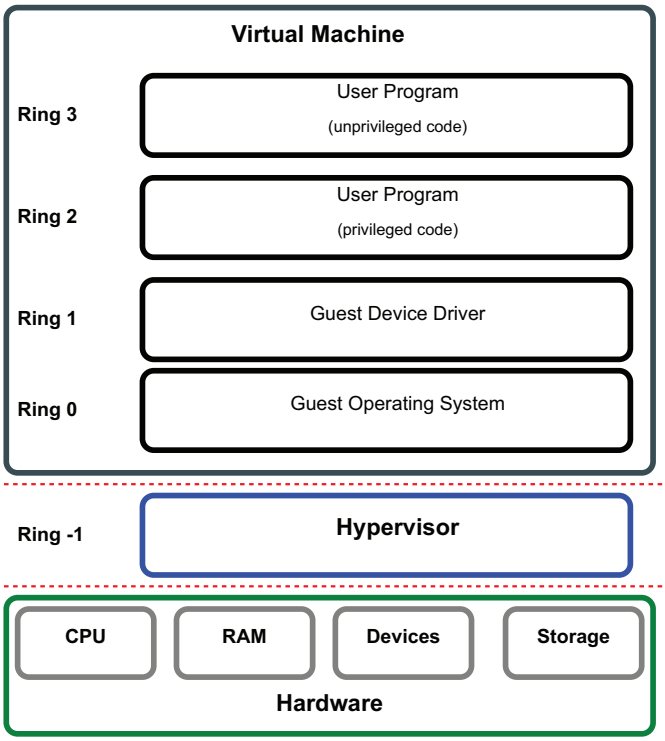

Fig. 3. Protection rings of hardware-assisted virtualization. The hypervisor has all the privileges over the physical hardware, while the guest operating system can directly run on the hardware with limitations defined by the hypervisor. All other rings require making system calls in order to execute a task on the hardware.

cally within the Docker platform and provides several features such as health checks, lifecycle management of individual containers, redundancy and failure handling in case of node failures, rolling software updates, and dynamic scaling of containers based on actual load. This tool is already included in Docker. Kubernetes, on the other hand, is a software solution for container orchestration typically used to manage a large number of containers on physical infrastructure. It offers additional features including service discovery and load balancing, storage orchestration, rollback and self-healing capabilities for container clusters, permissions management, and configuration management.

B. Model Deployment and Training

The integration of edge computing and cloud computing aligns with the requirements of Industry 4.0 [6]. However, distributed deep learning (DDL) involves network communication between compute nodes, which can slow down the DDL process [7]. Additionally, data exchange between cloud computing and edge computing is measured for billing purposes, which can be costly for small enterprises.

Another approach proposed in [8] involves the use of Kubernetes, Slurm, Ansible, and Docker to deploy a deep learning task on a cloud platform. While this approach is interesting, it requires the implementation of a complex local software infrastructure that is unnecessary for our use case.

Furthermore, maintaining such an infrastructure adds additional costs. Instead, we propose packaging the entire software stack in a container, eliminating the need for complex local infrastructure. Instead of using Slurm for load distribution, we suggest using the PyTorch API with MPI.

To overcome the need for secure physical HPC systems (which require privileges) when using MPI, one solution presented in [9] is to run a container with integrated MPI on a physical HPC infrastructure. However, this solution may not be applicable to small enterprises. Nevertheless, isolating the software stack in a container simplifies deployment, a practice commonly employed in HPC environments [10]. However, the approach proposed in [9] uses TensorFlow, which is known to be a slower framework than pyTorchk [11] and requires significant container image space.

An alternative to containerization is to utilize cloud services specifically designed for deep learning tasks [12]. This alternative also meets the requirements of Industry 4.0, particularly with the integration of IoT devices [13]. By leveraging a combination of edge computing, mobile computing, and cloud computing, energy can be saved and latency reduced. However, this solution can be costly. Moreover, cloud services dedicated to deep learning often have specific implementation requirements, and developers typically use Python for quick prototyping and local testing. As a result, their applications may not be compatible with the services provided by cloud platforms, necessitating additional time for code migration to a cloud service.

As a first step towards reducing costs and facilitating the deployment of Python deep learning applications, this paper conducts an analysis to assess the performance of a single virtual machine (VM) with multiple CPUs in executing a DL task but also on two and four VM. The study also involves measuring the behavior of the CPU, RAM, and disk I/O operations through the VMware interface. This analysis helps to establish the relationship between the number of CPUs and the required amount of RAM. Furthermore, the paper outlines the process of designing and compiling a lightweight container image from scratch, which encompasses the complete software stack necessary for implementing distributed deep learning (DDL). Finally, a cost estimation is presented, comparing the usage of multiple VMs with CPUs against a single VM with a GPU for accomplishing the same DL task. The findings indicate that a single VM with a GPU is approximately 30 times more cost-effective than employing four VMs with CPUs for achieving equivalent performance.

III. Proposed Work

The proposed architecture for industry 4.0 comprises two key components: the on-site manufacturing infrastructure and the cloud computing infrastructure. The on-site infrastructure encompasses production equipment equipped with sensors that collect and provide data to support a deep learning model. This model can be employed for various purposes, such as defects detection [14], predictive maintenance [15], optimizing production processes [16], reducing downtime [17], and improving quality control [18]. The deep learning model can be implemented centralized or replicated across multiple manufacturing devices to ensure real-time detection and decision making. In both scenarios, the architecture refers to the edge device, the physical machine, or the virtual machine that hosts the primary deep learning model as the *gateway device*. When replicas of the model are distributed among edge devices, the main model serves as the master model, while the replicas function as slave models. The slave models are updated following each update of the master model, facilitating efficient synchronization across the network. This approach simplifies the process of updating the entire set of replicas. The proposed architecture also considers deploying replicas using Docker container technology, which is feasible on edge devices like NVidia Jetson. By combining the capabilities of the on-site manufacturing infrastructure with the cloud computing infrastructure, the proposed architecture enables efficient data collection, analysis, and decision making in real time, contributing to the advancement of industry 4.0 applications.

The frequency of updating the main deep learning model in the proposed architecture varies depending on the specific use case and the volume of data involved. It can range from once a day to once a year, depending on the requirements. To facilitate the deployment and replication of the training script for the main model, the architecture utilizes Docker Swarm mode because of its simplicity and because it does not require additional piece of software. This allows for easy distribution of the script among a set of virtual machines. Many manufacturing facilities already possess physical servers that support virtualization, and they may also lease machines from cloud providers. By leveraging Docker Swarm mode, the proposed approach streamlines the deployment process by using a minimal container that includes the necessary software stack for training the deep learning model. This eliminates the need to package a complete GNU/Linux distribution within the container. The use of Docker technology ensures that the proposed approach remains independent of the underlying infrastructure. Furthermore, by utilizing a minimal container, the processing overhead is minimized, thus maintaining optimal performance. The lightweight nature of the container enables fast deployment, facilitating efficient and scalable model training. In the proposed architecture, all other required software components are assumed to be readily available within the virtual machine. It is assumed that the virtual machines run on a GNU/Linux distribution, which is commonly available as a standard template on cloud providers. This design choice simplifies the architecture and ensures that the necessary software components are easily accessible. An illustration of this architecture is provided in Figure 4, which highlights the key components and their relationships.

Several major cloud providers offer Kubernetes, while Docker Swarm mode is equally available since it is not dependent on any tools other than the Docker suite. Docker Swarm mode provides sufficient functionality to support the proposed architecture and is easy to implement. To ensure

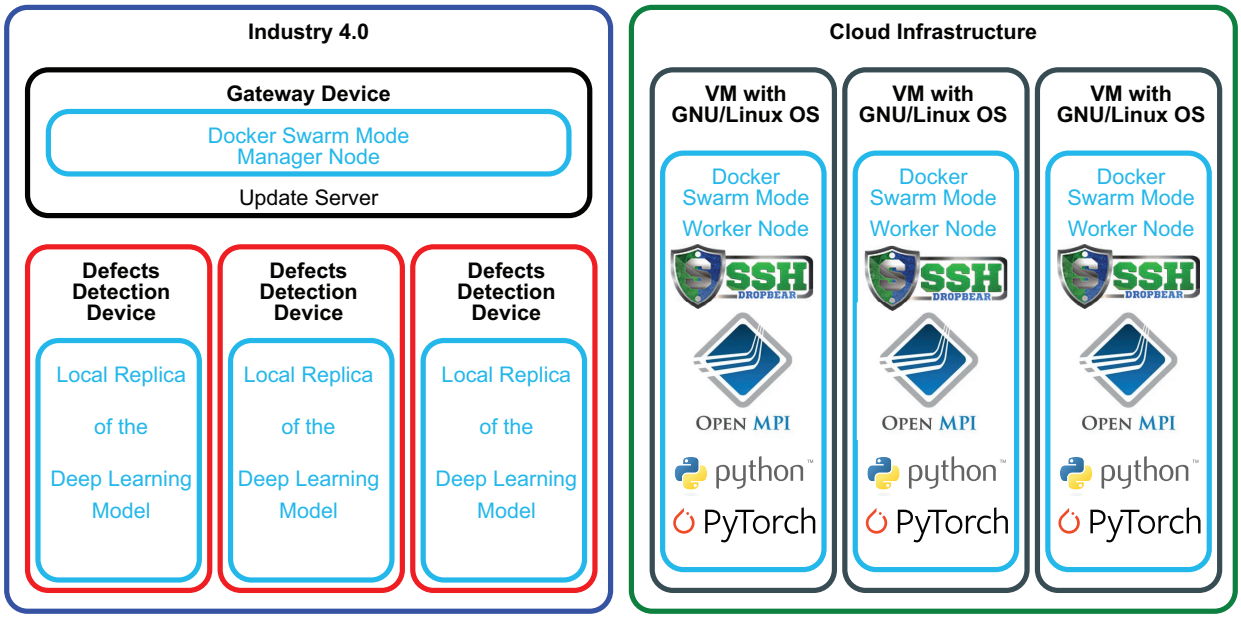

Fig. 4. High-Level Scheme of the Architecture. The left (resp. right) part in dark blue (resp. green) is the on-site (resp. cloud) infrastructure of Industry 4.0. Red shapes represent the devices that predict defects using the deep learning model. The gateway device, which acts as the Docker Swarm manager, is depicted as a black shape. In the cloud infrastructure, virtual machines are represented by dark gray shapes. All the Docker containers are shown as light blue shapes.

lightweight and fast deployment, containers dedicated to training DL models do not include an operating system. Instead, the proposed architecture leverages the operating system of virtual machines, allowing for hardware-assisted virtualization and minimizing software isolation through containerization. This approach also enables dense deployment and reuse of virtual machines that are already in use by companies.

IV. Experimentation

This section describes the design of experiments. The measurement of time was repeated 1000 times in order to ensure significant accurate values. Beyond 1000, it was not necessary because the largest deviation is always observed during the first iteration with a maximum of 61 seconds. All other deviations are under 5 seconds.

A. Distributed Deep Learning

In this paper, the industry 4.0 use case is to detect defects in printed circuit boards (PCB) [19]. Nevertheless, the convolutional neural network (CNN) and the preprocessing of data are adapted to be comparable to a method to distribute deep learning [20] on physical machines on CPU. Only 3 classes are considered: mouse bite, open circuit, and spur. Data augmentation [21] is also used to have a total of 6003 photos.

The DL task in this paper is to train the AlexNet CNN architecture [22] on the mentioned dataset. The required data preprocessing steps are as follows:

1) Image crop/scaling to 224×224;
2) Random horizontal flip transformation;
3) RGB-normalization with $\mu = (0.485, 0.456, 0.406)$, $\sigma = (0.229, 0.224, 0.225)$;
4) Conversion to a tensor data structure.

Other parameters are the cross-entropy loss function and the stochastic gradient descent with a learning rate of $\alpha = 0.001$ as an optimizer.

On a single VM, the pyTorch framework was used to apply parallelization with multithreading and multiprocessing. Each process had a local copy in the RAM of the DL model and loaded its own batch of data. This is called data parallelism. Between VMs, we also apply data parallelism with data replication on each VM. At each training step, a batch of data was used to feed the model and produce an output. The error was measured in the output to compute the gradient and update the model. At this step, updates were sent across all virtual machines to synchronize the model in all processes. This behavior was repeated until all data were used, which is called an epoch.

B. Container

The container images built with Docker technology were created in virtual machines. To avoid the replication of the whole GNU/Linux system inside the container, it was made from scratch. Like other frameworks, pyTorch requires a software stack to enable DDL. This is why each piece of software was compiled with a x86_64 architecture, mainly composed of OpenMP v5.0 for parallelization, OpenMPI 4.1.2 for distributed computing, DropBear SSH v2022.82 for communication between MPI application, Python v3.10.2, and pyTorch v1.12 because this is the fastest and most stable framework for training models [11]. The docker image is available at https://hub.docker.com/repository/docker/belegkarnil/

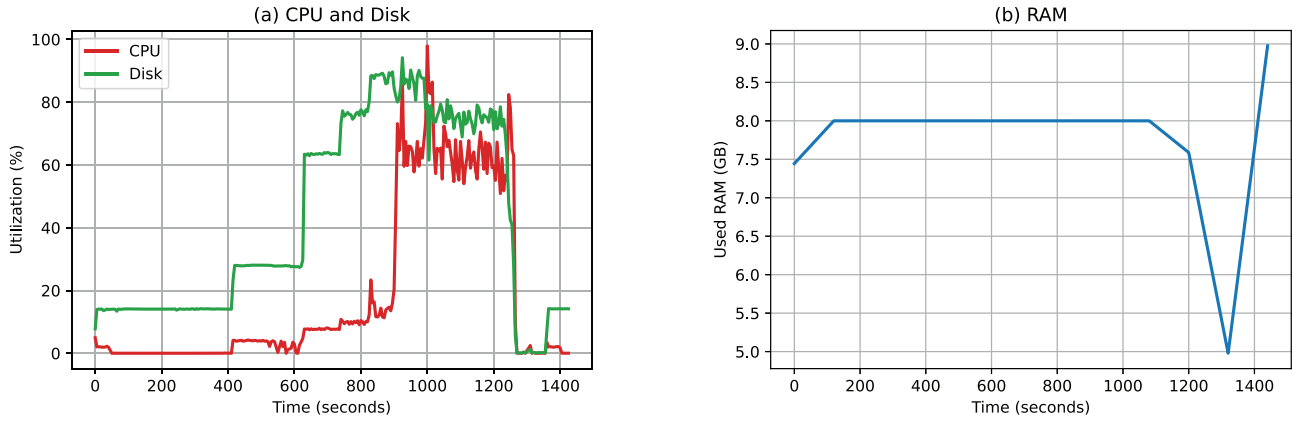

Fig. 5. Evolution of the use of resources on a single VM with 8 GB of RAM and 8 CPU cores. CPU and disk use are shown in Figure 5 (a) and the RAM use is shown in Figure 5 (b). Results were collected via the VMware interface.

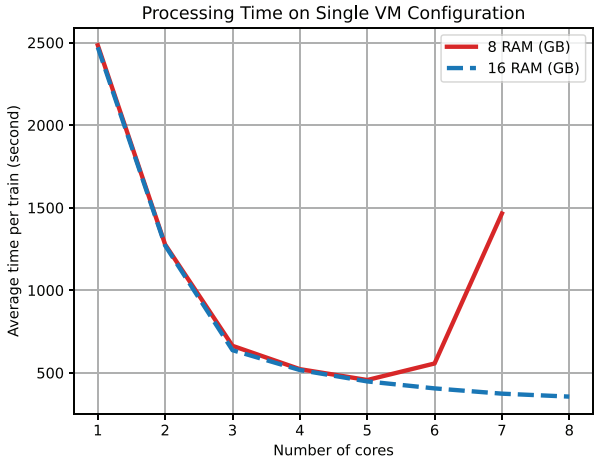

Fig. 6. Average time per train (second) required to execute the DL task depending of the number of the core. The red (resp. blue) line is the execution on a VM with 8 GB (resp. 16 BGB).

pytorch-cpu and was used as a parent image. The final compressed container had a size of 143.34 MB.

C. Dataset Deployment

Three approaches were used. The first approach involved building a new child Docker image by adding the dataset required for the DL task and the PyTorch script. Its compressed size is 275.36 MB. The second approach utilized a separate container with Samba to provide access to the data. Lastly, the third approach utilized a network file system (NFS) to provide access to the data.

V. Results

Due to the budget, it was not possible to run all possible configurations. Thus, this work focused on the variation in the number of CPU cores, as they are the compute resources. Figure 6 shows the average time per train required to execute the DL task on a single VM. A VM with only 8 GB was able to process our DL model and dataset up to seven cores. With eight cores, the amount of RAM was not sufficient to apply data parallelism on DL—that is, to replicate the model eight times and to load data in the memory. A VM with a larger amount of RAM (16 GB) was able to process and demonstrate experimentally that the more CPU cores there were, the less time it took to complete the task. On the other hand, the time increased when the number of cores increased and the amount of RAM was limited, as shown by the VM with 8 GB of RAM.

Figure 5 reports the CPU, RAM, and disk use over time on a single VM with eight CPU cores. The execution was stopped after 1435 seconds because the VM wanted to allocate more RAM than was available to run the DL task. From time 0 to around time 1000, the use of the CPU and the disk kept growing. This behavior corresponded to loading the DL model and a batch of data in memory for each CPU core. RAM was quickly and fully allocated at time 120. Then, at time 1320, all resources decreased and the memory was freed. Only 5 GB were allocated. According to the results obtained for the average time per epoch, this matched a new iteration of the epoch. Then, a new epoch started and the three metrics grew. The VM tried to allocate 9 GB of RAM but they were not available and the DL task crashed.

The DDL was applied to the DL task in order to execute it on two VMs and four VMs. The average time taken is reported in Figure 7. The speed achieved for the parallel DL task on a single VM was 1.7 times faster than that for DDL on two VMs. The four VM setup executed the DDL task 3.4 times faster than the parallel DL task on a single VM. The setups on VMs with 8 GB of RAM were not executed with more than five CPU cores because of the results that showed an increase in time.

The results showed that deploying a DDL task on VMs can reduce the compute time required. A linear speedup factor of up to four VMs could be achieved. We therefore believe that with more VMs, it would be possible to speed up the

TABLE I
AMAZON EC2 ON-DEMAND PRICES OF VM TEMPLATES.

	GPU	CPU			
Instance name	p4d.24xlarge	c6g.medium	c6g.large	c6g.xlarge	c6g.2xlarge
Compute resource	8 GPU	1 vCPU	2 vCPU	4 vCPU	8 vCPU
Price per hour	32.7726 USD	0.0340 USD	0.0680 USD	0.1360 USD	0.2720 USD

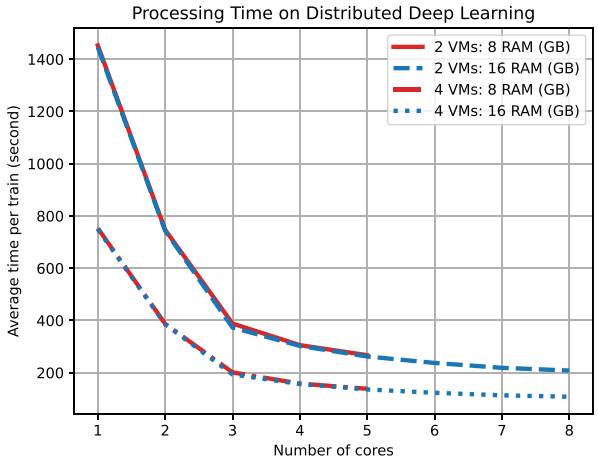

Fig. 7. Average time per train (second) required to execute the DDL task depending of the number of cores and VMs. The red (resp. blue) line shows the execution on a VM with 8 GB (resp. 16 BGB). The solid lines show the execution on 2 VMs and the dashed lines show the execution on 4 VMs.

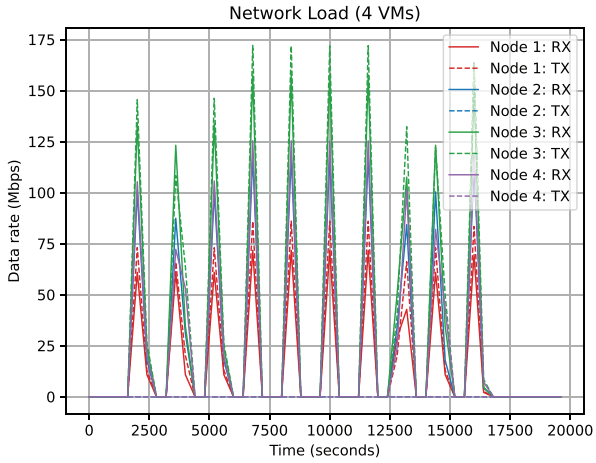

Fig. 8. Data rate for each of the 4 VMs (red, blue, green and purple). Transmit (TX) and receive (RX) are respectively in solid and dashed lines. Data point are extracted each 20 seconds.

compute time. Nevertheless, at a certain point, the network communication will become so intense that the speedup factor will decrease. This is the cost of communication.

In all the setups, the training time for the DL model was less than one hour. However, it should be noted that these results may vary depending on the complexity of the model and the size of the dataset. In the DDL setup, the location of data using the Samba approach introduces an average additional time of 0.28 seconds per epoch compared to the baseline, while the NFS approach introduces an average additional time of 0.19 seconds per epoch.

Figure 8 reports the network load of the least favorable network load, i.e. when network traffic is higher. The maximum transmission rate (Tx) is 172Mbps and the maximum reception rate (RX) is 157Mbps. The physical infrastructure is able to handle the load. As a result, network traffic has not slowed down the DDL training process.

Amazon AWS EC2 provides VM templates, and several of them are listed in Table I. Among these templates, the `x6g.2xlarge` template with 8 CPUs was suitable for our setup, priced at 0.2720 USD per hour. With four VMs, the total price becomes 1.088 USD per hour, compared to 32.7726 USD per hour for a GPU-capable VM. The method proposed in this paper is 30 times cheaper than using a virtual machine with a GPU. Nevertheless, GPUs can accelerate the training with a speedup of 11.79 [20]. On average, over several hours of calculations, using several VMs on CPU is 2.5 times cheaper.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we showed that it is possible to easily deploy a distributed deep learning application containerized with Docker Swarm mode. Nevertheless, it is quite important to choose a VM configuration where the batch processing and data replication can fit in the RAM once per core. We also experimentally demonstrated that using multiple on-demand cheap VMs, each with several vCPUs, enables one to quickly execute a deep learning task. Finally, we provided a small base image for DDL which is publicly available.

In future works, we are interested in deploying this methodology on public cloud providers such as Amazon AWS, Google Cloud, and Microsoft Azure. Considering these providers allows us to compare the performances of the CPU versus GPU versus Tensor Processing Units (TPU). Moreover, we want to explore how the source of data affects the results and if our methodology can also be successfully applied with GPUs.

REFERENCES

[1] J. Villalba-Diez, D. Schmidt, R. Gevers, J. Ordieres-Meré, M. Buchwitz, and W. Wellbrock, "Deep learning for industrial computer vision quality control in the printing industry 4.0," *Sensors*, vol. 19, no. 18, p. 3987, 2019.

[2] K. Demertzis, L. Iliadis, N. Tziritas, and P. Kikiras, "Anomaly detection via blockchained deep learning smart contracts in industry 4.0," *Neural Computing and Applications*, vol. 32, no. 23, pp. 17 361–17 378, 2020.

[3] V. Hegde and S. Usmani, "Parallel and distributed deep learning," in *Tech. report, Stanford University*, 2016, pp. 1–8.

[4] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[5] C. Anderson, "Docker [software engineering]," *Ieee Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[6] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*. IEEE, 2017, pp. 328–339.

[7] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, "Sparse binary compression: Towards distributed deep learning with minimal communication," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[8] R. Doukha, S. A. Mahmoudi, M. Zbakh, and P. Manneback, "Deployment of containerized deep learning applications in the cloud," in *2020 5th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech)*. IEEE, 2020, pp. 1–6.

[9] D. Brayford, S. Vallecorsa, A. Atanasov, F. Baruffa, and W. Riviera, "Deploying ai frameworks on secure hpc systems with containers." in *2019 ieee high performance extreme computing conference (hpec)*. IEEE, 2019, pp. 1–6.

[10] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Sarus: Highly scalable docker containers for hpc systems," in *International Conference on High Performance Computing*. Springer, 2019, pp. 46–60.

[11] J.-S. Lerat, S. A. Mahmoudi, and S. Mahmoudi, "Single node deep learning frameworks: Comparative study and cpu/gpu performance analysis," *Concurrency and Computation: Practice and Experience*, p. e6730, 2021.

[12] S. Boag, P. Dube, K. El Maghraoui, B. Herta, W. Hummer, K. Jayaram, R. Khalaf, V. Muthusamy, M. Kalantar, and A. Verma, "Dependability in a multi-tenant multi-framework deep learning as-a-service platform," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 43–46.

[13] H. Wu, Z. Zhang, C. Guan, K. Wolter, and M. Xu, "Collaborate edge and cloud computing with distributed deep learning for smart city internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 9, pp. 8099–8110, 2020.

[14] D. Powell, M. C. Magnanini, M. Colledani, and O. Myklebust, "Advancing zero defect manufacturing: A state-of-the-art perspective and future research directions," *Computers in Industry*, vol. 136, p. 103596, 2022.

[15] M. H. Abidi, M. K. Mohammed, and H. Alkhalefah, "Predictive maintenance planning for industry 4.0 using machine learning for sustainable manufacturing," *Sustainability*, vol. 14, no. 6, p. 3387, 2022.

[16] C. Zhou, W. Wang, Z. Hou, and W. Feng, "Milling cutter wear prediction based on bidirectional long short-term memory neural networks," in *ISMSEE 2022; The 2nd International Symposium on Mechanical Systems and Electronic Engineering*. VDE, 2022, pp. 1–6.

[17] E. Balamurugan, L. R. Flaih, D. Yuvaraj, K. Sangeetha, A. Jayanthiladevi, and T. S. Kumar, "Use case of artificial intelligence in machine learning manufacturing 4.0," in *2019 International conference on computational intelligence and knowledge economy (ICCIKE)*. IEEE, 2019, pp. 656–659.

[18] J. Breitenbach, I. Eckert, V. Mahal, H. Baumgartl, and R. Buettner, "Automated defect detection of screws in the manufacturing industry using convolutional neural networks," in *Proceedings of the 55th Hawaii International Conference on System Sciences*, 2022.

[19] P. Wei, C. Liu, M. Liu, Y. Gao, and H. Liu, "Cnn-based reference comparison method for classifying bare pcb defects," *The Journal of Engineering*, vol. 2018, no. 16, pp. 1528–1533, 2018.

[20] J.-S. Lerat, S. A. Mahmoudi, and S. Mahmoudi, "Distributed deep learning: From single-node to multi-node architecture," *Electronics*, vol. 11, no. 10, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/10/1525

[21] W. Huang and P. Wei, "A pcb dataset for defects detection and classification," *arXiv preprint arXiv:1901.08204*, 2019.

[22] 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.