arXiv:2208.02498v2 [cs.DC] 14 Nov 2022

A Container-Based Workflow for Distributed Training of Deep Learning Algorithms in HPC Clusters

Jose González-Abad[1*], Álvaro López García[1†] and Valentin Y. Kozlov[2†]

[1*]Instituto de Física de Cantabria (IFCA), CSIC-Universidad de Cantabria, Santander, Spain.
[2]Karlsruhe Institute of Technology (KIT), Steinbuch Centre for Computing (SCC), Karlsruhe, Germany.

*Corresponding author(s). E-mail(s): gonzabad@ifca.unican.es;
Contributing authors: aloga@ifca.unican.es;
valentin.kozlov@kit.edu;
†These authors contributed equally to this work.

Abstract

Deep learning has been postulated as a solution for numerous problems in different branches of science. Given the resource-intensive nature of these models, they often need to be executed on specialized hardware such graphical processing units (GPUs) in a distributed manner. In the academic field, researchers get access to this kind of resources through High Performance Computing (HPC) clusters. This kind of infrastructures make the training of these models difficult due to their multi-user nature and limited user permission. In addition, different HPC clusters may possess different peculiarities that can entangle the research cycle (e.g., libraries dependencies). In this paper we develop a workflow and methodology for the distributed training of deep learning models in HPC clusters which provides researchers with a series of novel advantages. It relies on udocker as containerization tool and on Horovod as library for the distribution of the models across multiple GPUs. udocker does not need any special permission, allowing researchers to run the entire workflow without relying on any administrator. Horovod ensures the efficient distribution of the training independently of the deep learning framework used. Additionally, due to containerization

and specific features of the workflow, it provides researchers with a cluster-agnostic way of running their models. The experiments carried out show that the workflow offers good scalability in the distributed training of the models and that it easily adapts to different clusters.

Keywords: Distributed Training, Deep Learning, High Performance Computing, udocker, Docker, Horovod

Declarations

- Conflict of interest/Competing interests (check journal-specific guidelines for which heading to use): Not applicable
- Ethics approval: Not applicable
- Consent to participate: Not applicable
- Consent for publication: Not applicable
- Availability of data and materials: Data and materials are available, URLs are provided through the manuscript
- Code availability: Code is available, URLs are provided through the manuscript
- Authors' contributions: All authors contributed equally to the study conception and design of the workflow. All authors read and approved the final manuscript.

1 Introduction

The machine learning subfield known as deep learning [1] has been the protagonist of a revolution in recent decades. The growing amount of data available as well as the increase in the computing power of current hardware has allowed neural networks algorithms to become state-of-the-art (SOTA) in multiple complex tasks such as language processing, computer vision and image generation. These successes have led researchers in more specific scientific areas to

begin applying these techniques in their own fields. For this reason, these algorithms can now be seen applied in fields as diverse as particle physics [2, 3], cosmology [4, 5], biology [6] or climatology [7, 8], just to cite some examples.

Given their nature, this kind of algorithm is very resource-intensive. The most recent SOTA results have been obtained through large clusters, which are generally provided with specialized hardware targeted to accelerate the execution of these algorithms. Specifically, in deep learning, graphical processing units (GPUs) have shown the greatest speedup. This has led to the development of powerful clusters equipped with multiple GPUs communicating between each other. Usually, researchers get access to these machines in order to run their experiments, but installing and configuring the required software to train a deep learning model in these multi-user environments is not an easy task, mainly due to restrictions in permissions. Also, the deployment of models in such distributed environments requires advance knowledge in computing, which applied scientists may not have. There are cloud technologies with simple interfaces that seek to reduce this barrier [9], but due to particular requirements of academia and research (e.g. high throughput and low latency networks) it is still common to access High Performance Computing (HPC) clusters.

One way to alleviate these problems is by allowing users of HPC clusters to use containerization tools to encapsulate and run specific software stacks. This kind of tools allows the isolation of software from the rest of the system through the use of specific features of the Linux kernel (e.g., kernel namespaces and Linux controls groups). These solutions are very popular and have been widely adopted in both industry and academia. In the former, they have become an important part of the development and deployment of applications and services, while in the latter they have helped increase the portability of the software required to run an experiment, as well as ease its reproducibility.

The most widespread containerization tool is Docker [10], however, given the escalation of privileges needed, it is not suitable for multi-user environments such as HPC clusters. Due to this, new containerization tools focused on running in such environments began to be developed, among which Singularity [11], Charliecloud [12] and Shifter [13] stand out. Some of these tools, while allowing non-root users to execute their containers, still need root permissions for the proper installation and configuration of the tool. Some others avoid this by recurring to specific features of the Linux kernel.

The tool used in the experiments described in this paper is udocker [14]. It is a user tool oriented to the execution of Docker containers without the need for root permissions or any other kind of privileges. This tool supports a large subset of Docker commands, so the researcher only needs to know a few notions of the latter. Given udocker's sole dependency on Python, the researcher hardly needs any administrator intervention for installing it, in contrast with other solutions. In this way, users may safely leverage new libraries and frameworks without recurring to HPC administrators, making the research cycle more dynamic.

Running scientific experiments as containers in an HPC cluster is not trivial. Sometimes the code needs to be adapted and special versions of software may need to be installed in order to take advantage of specialized hardware. For this reason, several workflows have recently been developed [15–17] to combine in a single flow the entire process required to run an experiment, from the development of the experiment to its deployment on a cluster, using the previously mentioned containerization tools.

In this paper, we develop a container-based workflow to train deep learning models in multiGPU environments in HPC clusters. To the best of our knowledge, this workflow is the first one specifically oriented to train this kind of models on specialized hardware in a distributed manner while filling previous research gaps. This workflow provides HPC users with a series of novel advantages:

- No administrator or superuser intervention is needed, accelerating the research cycle.
- Cluster-agnosticism, which allows researchers to run their experiments in any x86-64 architecture cluster available with minor changes.
- Efficient scaling of the training of the model with respect to native setups.

For this, we use udocker as containerization tool and Horovod [18] as library for distributing the training of the model across GPUs. Its use allows the distribution of the training independently of the framework used (TensorFlow, Keras, Pytorch and MXNet). The conjunction of these tools, developed integration efforts and good practices conform the workflow presented in this work. As a result, we contribute with a novel workflow built on the difficulties that researchers face when training deep learning models on multiGPU HPC clusters. Furthermore, we show these advantages by running different synthetic and real experiments on different clusters, without requiring administrator intervention.

The paper is structured as follows. Section 2 introduces all the technologies involved in the workflow. Section 3 introduces related work and shows how our workflow fills the gaps in the literature. In Section 4 the actual workflow is presented, while Section 5 describes the experiments. Section 6 details the systems where these experiments were executed, and Section 7 shows their results. Finally, Section 8 contains the discussion of these results and Section 9 the conclusions of the work.

2 Background

2.1 Distributed Deep Learning

A GPU used to be enough to train a deep learning model. It allowed researchers to speed up the training process and therefore the experimentation with respect to a standard central processing unit (CPU). Due to the increase in the

complexity of these kinds of models and the datasets used to train them, environments with just a GPU started to introduce limitations caused by its low available memory and increased training time.

As a consequence of these complexities, researchers had to start using distributed infrastructures to train their models. These infrastructures are usually made up of several computing nodes, each of which is equipped with one or more GPUs. This heterogeneity brings several distributed computing concepts to take into account, such as communication between nodes, storage management and effective use of computing resources. This fact entangles researchers' work, since these infrastructures are often necessary to achieve SOTA in deep learning [19, 20].

The two most commonly used parallelization strategies for this kind of models are data parallelism and model parallelism [21]. In the former, each worker (GPU in this case) loads a copy of the model and computes the gradients of a chunk of training data. These gradients are then averaged and used to update each of the models. The difficulty of this strategy lies in the synchronization of the parameters between workers. In the latter, the model is split between workers, so each of them computes the correspondent part of the forward and backward pass. This split is not trivial and is determined by the model itself. Considering its flexibility and ease of use regarding model parallelism, data parallelism is more widely used by researchers.

There are many deep learning frameworks available, each with its characteristics and syntax [21]. Based on the framework, the parallelization strategy may vary, adding another layer of complexity to the experimentation process. Tensorflow [22] supports both kinds of strategies, focusing on data parallelism. Depending on the selected strategy, it is possible to use different synchronizations between workers (parameter server and all-reduce algorithms). All these strategies are also accessible within Keras [23] through the `tf.Keras` API. In the same way, Pytorch [24] offers a variety of strategies based on data parallelism in addition to some strategies focused on more specific scenarios like RPC Distributed Training.

The variety of deep learning frameworks and their implementations of distributed strategies complicates the task of researchers without advanced knowledge in distributed systems since they must first know which strategy is the best for them in terms of efficiency and scalability with respect to their model, know extremely well the syntax of the framework they are using and properly adapt the code for the distributed execution. As a result of the need to simplify this process, Horovod was born.

2.1.1 Horovod

Advances in deep learning made large companies like Uber find it necessary to scale their models to multiGPU environments. They began by testing Tensorflow, the most widespread framework at that time, but they ran into some issues. The first was its steep learning curve due to the high number of concepts

necessary for the correct implementation, the second was the poor scalability achieved. These facts led Uber to develop Horovod [18], a distributed deep learning training framework built over deep learning frameworks like Tensorflow, Keras and Pytorch. Twin pillars of Horovod are:

- **Ease of use**: Regardless of the framework we use, with Horovod we only need to add a few lines of code to train our model in both GPU and multi-GPU environments. This removes barriers derived from differences between syntax.
- **Scalability**: Its use ensures good scalability thanks to its implementation based on various communications standards [18].

All communication between workers is done using a collective communications library. Horovod allows the use of either a Message Passing Interface (MPI) implementation like OpenMPI or Gloo, a Facebook library oriented to communicate workers in machine learning applications. These libraries are in charge of the synchronization of the parallel processes and allow the user to correctly manage the execution.

The distribution of the training could be done with one of these libraries, but operations like gradients average among workers would introduce bottlenecks in the process. Horovod recently added the NVIDIA Collective Communication Library (NCCL) to its framework. This library includes communication routines optimized for NVIDIA GPUs that speed up data exchange. Among those included, the ring-Allreduce based on an implementation inspired by [25] stands out. In this algorithm, each worker communicates with two others. During these communications, the workers exchange data and perform reduce-like operations. This process is repeated until the complete reduction of the data.

At present, Horovod combines MPI or Gloo for CPU oriented operations and process management and NCCL for GPU communications. All these features make Horovod an appropriate library for the distributed training of deep learning algorithms.

2.2 HPC Clusters

In research and academia, scientists normally gain access to GPUs through HPC clusters. These are usually managed by computing centers specialized in this type of technology that offers their computing power as a service to a wide range of researchers.

These computing clusters, which typically run under a Linux operating system[1], are composed of a collection of different nodes connected via high throughput and low latency interconnects like Infiniband[2]. In this way, it is possible to parallelize tasks between nodes and process large amounts of data efficiently. In some of these clusters, several of their nodes are provided with a

---

[1]https://www.top500.org/statistics/details/osfam/1/
[2]https://www.infinibandta.org/

GPU besides the CPU. Through access to several of these nodes, it is possible to distribute the training of deep learning models.

Once researchers have gained access to an HPC environment, they connect remotely and execute their workloads through a batch system or job scheduler such as Slurm [26]. For the proper execution of the job, it is necessary to correctly configure the environment in which it will run. This mainly includes the configuration and installation of programming tools and libraries. In jobs that need to be run on GPUs, this task becomes even more complicated since they depend on more complex applications linked to the GPU like the Compute Unified Device Architecture (CUDA) or the NVIDIA CUDA Deep Neural Network library (cudNN). Because deep learning frameworks depend on these applications, their installation must be done taking into account the versions of these available in the HPC cluster.

2.3 Containerization technologies

When a new user accesses an HPC facility, the first thing they need to do is set up their own environment. In these systems user permissions are restricted and the computing environment (i.e., operating system and libraries) is fixed, which can lead to issues in basic aspects like software management (e.g., no package manager available) or no administrative privileges, limiting the usage of certain software. These problems are common in computational science and are present when the user is forced with a fixed environment [27]. Moreover, once the setup is done at one facility, it may not correctly adapt to the heterogeneity of systems that researchers can manage, since if they want to deploy the same environment in another HPC they need to redo the installation with the difficult additions of doing it on a different system. They may also find differences between the local system on which they programmed the application and the HPC on which they deploy it. These issues add another layer of complexity to the research process.

One solution to these problems is containerization. This technology allows all the software, settings, code and environment variables of a user to be packaged in what is known as containers, then this software stack can be deployed on any machine with ease. Unlike virtual machines that virtualize at the hardware level through a hypervisor, containers virtualize at the operating system (OS) level using several features of the Linux kernel. Containers are isolated processes running on top of the OS kernel, so there is no need for hardware virtualization. These features make containers more efficient than Virtual Machines [28].

Specifically, in HPC environments, the use of containers can make the development and deployment process easier for researchers. First, they can stack their environment in an isolated file, which allows them to take better control of aspects like the libraries used and their versions. This stack allows them to run the same both in their local environment and in HPC ones, avoiding tedious installations. In addition, it also facilitates reproducibility since this file can include everything necessary to run these experiments (e.g., code, models,

seeds). The latter plays a fundamental role in the field of deep learning, where the exact reproducibility of the results does not depend solely on the code. We briefly present some of the most popular containerization technologies in what follows.

2.3.1 Docker

Docker [10] is the most popular container creation and execution solution. Its implementation is based on two features of the Linux kernel: `namespaces` and `cgroups`. The former is responsible for building the layer of isolation, making the container processes run isolated from the rest of the host. The latter is in charge of managing and limiting the hardware resources assigned to the containers. In addition, Docker also allows the distribution of containers thanks to DockerHub, a registry service focused on sharing container images. Unfortunately, this solution has a major drawback in our context, which is the dependency on a root owned daemon process, requiring, therefore, administrative privileges to run a container. In multi-user systems like HPC clusters, a standard user does not have these privileges, making Docker unsuitable for these types of environments.

2.3.2 Singularity

Singularity [11] is a tool that offers image-based containers which allow software encapsulation and its execution in HPC environments, given its non-root execution. It is based upon `setuid` file permission, which allows running a program with escalated permissions. A `setuid` binary gives root access to certain basic operations required for the proper execution of the container. This method is flexible but dangerous from a security perspective. It is also possible to run Singularity in a `non-setuid` mode, but it requires unprivileged user namespaces, a fairly recent Linux kernel feature that may not be available on all HPC environments. Moreover, apart from the security implications, the main issue with `setuid-mode` is that the binary needs to be previously installed with root permissions, so the intervention of an administrator to install and configure this tool is needed beforehand.

2.3.3 udocker

udocker [14] is a tool that enables the execution of containers without requiring administrative privileges. It focuses on the execution of containers in multi-user environments, leaving the process of creation and distribution of images to Docker. In order to run these Docker containers, it is necessary to implement chroot-like functionality without the need for root access. This is achieved in three different ways: unprivileged user namespaces, `PTRACE` mechanism, and `LD_PRELOAD` mechanism. The first works the same as in Singularity. The second allows changing the pathnames of the system calls in runtime, making them reference pathnames within the container. The third allows the overriding of shared libraries, which allows running those of the chroot-environment (see

[14] for more details). The required binaries for these operations are statically compiled, which increases portability. udocker does not require any restrictive permissions beyond the user's own, eliminating the need for administrative privileges required by other tools for its installation and execution. This fact also provides more autonomy to users, since they do not need to rely on the administrator's intervention.

3 Related Work

The need to run experiments in multi-user environments has led to the development of workflows oriented to such purpose. In [9] a set of tools and services to cover and ease the whole machine learning development cycle is presented, although useful it focuses on cloud computing and not on HPC clusters. Regarding these environments, the increased efficiency and lack of overhead of containers compared to virtual machines [29] has led to the development of container-based workflows.

In [30] the use of Docker containers in HPC clusters is studied. They compare the deployment of applications with Docker containers and virtual machines and conclude that the former has more advantages in addition to no overheads. In [31] authors extend the Docker platform to meet the requirements of HPC systems and ease the execution of workflows on such environments. In [32] a Docker's wrapper is developed to execute containers via the Slurm Workload Manager in the cluster of the University of Oslo. These workflows rely on Docker, so it is still necessary for the cluster to already have this system installed or for the user to ask the administrator to install it. This dependency makes these workflows non-cluster-agnostic, since the user can not extend them to other clusters without the respective administrators' intervention.

In parallel, researchers developed workflows with the same objectives but using containerization tools focused on HPC environments. In [15] a workflow based in Singularity is proposed for the deployment of containers developed in a local environment both to the cloud and to an HPC cluster. In [33] authors develop a workflow that allows users to run containers in a cluster using Shifter [13], a software oriented to running containers in multi-user environments. Although these workflows are well adapted to multi-user environments, they still depend on an administrator since Singularity needs a daemon with root permissions and Shifter needs to be installed on all nodes where the experiment is run. Furthermore, these works still focus on a single cluster.

The developments presented so far are for general purpose experiments. Deep learning experiments have their own characteristics that make them even more complex to run in HPC contexts (e.g., requiring specific libraries to control GPUs). In [16] authors focus on applications oriented to deep learning, for which they use Charliecloud [12]. Subsequently, a similar workflow is applied to the training of a deep learning model in the field of particle physics [17]. The developed workflow allows them to take advantage of the hundreds of CPUs available in their cluster. These workflows, although they focus on deep

learning algorithms, do not use specialized hardware (GPUs) which usually requires more complex configurations. Furthermore, many of these workflows are still focused on a specific cluster, therefore not being cluster-agnostic.

In [34] a study of the scalability of udocker with respect to the number of GPUs used to train a deep learning model is presented. udocker performance is compared against the native environment and other containerization tools, and it is observed that it does not present a significant overhead in multiGPU scenarios.

In addition to the containerization tools explored in Section 2.3, there are others that seek to fulfill the same purposes. Podman [35] is another container-ization tool with features similar to those of Docker, which also offers the possibility of running containers without requiring root permissions. The disad-vantage of such tools with respect to the developed workflow is that they either require administrator permissions for installation or, for compilation, depend on software that may not be available in all clusters (e.g. Podman requires Go[3] for building it from source). udocker installation can be performed by the user and its main dependency is Python, the current most popular programming language. Other tools aim to facilitate the management and installation of libraries in multi-user spaces, for example Spack [36], a package manager ori-ented to these environments. Through a simple syntax, it allows users a better management of versions and configurations options in HPC clusters. However, it does not have all the benefits that a containerization tool can offer in the context of this work (see 2.3).

Recently, work has been developed that explores the future difficulties that researchers will face when scaling up their experiments on HPC clusters. In [37] the difficulties involved in moving workflows from CPU-based to GPU-based systems are addressed. Authors propose initial steps towards a framework for the automatic deployment of optimized containers in such systems. In [38] authors provide an analysis of the challenges in providing portability and repro-ducibility to containers in HPC. One of these challenges is the need of mapping specific host libraries to the container.
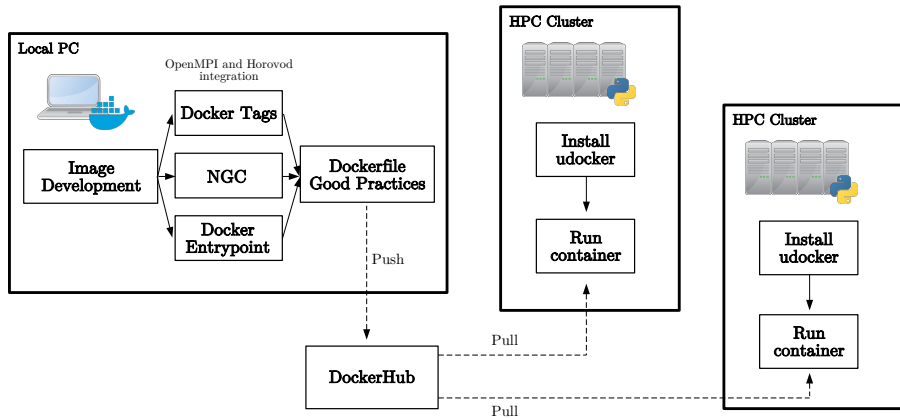
In this work we develop a new workflow offering a number of advantages over more conventional ones. It is based on udocker, so the user does not need administrator intervention to run their containers, what is more, this allows them to run their containers in a cluster-agnostic way since users can manage its installation relying on a very basic set of dependencies. The independence that this workflow gives to the user (no administrators' intervention required) allows for an increase in the speed of the research cycle. Previous work has explored the performance of udocker, and it has been seen that scalability is not harmed, thus being able to easily scale through various GPUs. All this makes udocker the perfect candidate to develop workflows oriented to the training of deep learning models in multi-user multiGPU environments in a cluster-agnostic way.

---

[3]https://go.dev/

4 Methodology and Workflow

Given the aforementioned necessity of training deep learning models in distributed environments, we have developed a methodology focused on this task when some HPC cluster is involved. It is based mainly on the interaction between three tools: Docker, udocker and Horovod. Figure 1 shows a diagram of the workflow.

Fig. 1 Diagram of the developed workflow

Under this methodology, researchers develop the image of their local environment with Docker. They can start from images already built and tested by the community to support computation on GPUs. Once they have an image of their experiment's environment, they upload it to DockerHub making it accessible through the Internet. When they want to deploy this image in any HPC cluster, they just need to install udocker in the cluster and download the image from DockerHub. Horovod is in charge of distributing the model training among the selected nodes of the cluster without harming scalability.

In this section, we explore this workflow and study the different problems that arise and how to tackle them.

4.1 Image Development

Docker provides virtualization through running images as containers. An image is a file that contains all the code, libraries, tools and dependencies that define the environment. A container is a virtualized runtime environment responsible for isolating the execution of the image from the rest of the environment. The image building process is defined in a file known as Dockerfile.

A Dockerfile is a file belonging to the Docker environment in which all the commands used to build the image are defined as a series of steps. When the image is built, these commands are executed sequentially. Each command generates a layer that is stacked on top of the ones generated with previous

commands. In this way, the final image is conformed of a set of read-only layers. When a container of a certain image is executed, Docker stacks a new read-write layer that allows to work on top of the previously installed ones.

Setting up working environments with GPU support is not a trivial task. Correct installation and configuration of GPU-related libraries such as CUDA, cudNN and NCCL are required. This task is complicated in most HPC environments given restrictions on user permissions and dependency on already installed software. Thanks to the Docker's layer structure and the available already configured images (e.g., TensorFlow, NVIDIA), it is possible to avoid this tedious configuration, since we can stack custom layers over those already supporting the needed GPU libraries.

Horovod allows researchers to abstract from the computational complexities involved in the adaptation to training in distributed environments. Its use does not limit researchers in any way, since it can be used with the most popular deep learning frameworks with minor changes in the syntax, which allows the standardization of the training with respect to the framework used.

Once this basic software is installed, researchers can move on to installing more specific things, such as the libraries or additional programming languages that their experiments require. During this process they can define the versions of these, ensuring an environment fully compatible with their experiment.

This process can be carried out in its entirety in the local environment, which allows researchers to test the developed environment easily and safely. Docker also has instructions like COPY that allow copying local files to the image. This allows researchers to include code in the image so that not only they can have their entire environment in a single file, but also the code to reproduce their experiment. In this way, researchers can easily develop a fully reproducible experiment while working on it. This removes well-known problems as libraries version mismatches or compatibility issues, since all the elements required to reproduce the experiment are encapsulated in the same environment.

Once researchers have defined the Dockerfile, they own a file that contains the specific environment of their research, from OS to high level libraries passing through specific dependencies and GPU-related software. For example, a general image with GPU-support, OpenMPI, and Horovod could be built and shared among researchers. Taking this image as a basis, other researchers would only need to add their code and specialized libraries to train deep learning models in any HPC cluster under the workflow developed in this work.

4.2 OpenMPI and Horovod integration

Parallel computing systems such as HPC clusters need message passing approaches to communicate the different nodes. MPI is the de facto standard when it comes to message passing. There are many implementations of this standard, OpenMPI [39] being the most widespread. OpenMPI is an open-source MPI implementation that enables efficient communications

between applications on different nodes and supports advanced protocols such as Infiniband.

Horovod coordinates work between processes via MPI, using OpenMPI as the basis for `horovodrun`, its MPI wrapper. In addition, MPI is also needed to communicate the different udocker containers distributed among nodes. Due to inconsistencies between some commands, it is necessary that the version of OpenMPI available in the cluster and that of the udocker container match. This fact introduces difficulties in the workflow, since now researchers depend on the version of OpenMPI installed in the cluster in which they run the experiment, making the process non-cluster-agnostic. We propose several solutions to this issue.

4.2.1 Docker Tags

The most straightforward solution would be to build the image taking into account the version of OpenMPI available in the HPC in which we want to run the experiment. The main problem is that this solution complicates the adaptation of the workflow to various HPC environments.

One way to tackle this problem is by using Docker tags. Tags are names used to group an image with respect to, for example, some version or variant. In this case, researchers could, from their base image, upload variants to DockerHub according to the OpenMPI version installed and classify them based on tags. Hence, they can choose which tag to pull based on the OpenMPI version available in the HPC cluster in which they plan to run the experiment.

4.2.2 NGC catalog

NVIDIA GPU Cloud[4] (NGC) is a catalog of GPU optimized software developed by NVIDIA. It offers images already built and optimized for the training of deep learning models in multiGPU environments. Its objective is to provide researchers with ready-to-run images which they can pull in any environment, thus avoiding complex installations and configurations.

The images it offers are optimized and support the latest NVIDIA hardware. They can be classified based on the included software and applications. It also offers software development kits (SDKs) for GPU computing in HPC clusters. Some of these images come with Horovod and OpenMPI integrated, so they can be used to train models in multiGPU environments within our developed workflow as long as these versions match the ones available in the HPC cluster.

NVIDIA offers this as already built images, so researchers have less control over the software installed on them. This can lead to incompatibilities between the included software and the one researchers need to install, making proper configuration difficult. Furthermore, the software included in this catalog is mainly optimized for GPUs with an architecture equal to or more recent than Pascal.

---

[4]https://catalog.ngc.nvidia.com/

4.2.3 Docker Entrypoint

When a container is executed, Docker runs a command that can be defined in the Dockerfile through the ENTRYPOINT instruction. This command is executed whenever the container is run. It is possible to pass arguments to this command through the CMD instruction. These parameters can be overwritten when we run the container, an example is shown in Fig.2.

```
1        ENTRYPOINT ["/bin/bash", "-c"]
2        CMD ["ls"]
```

Fig. 2 Dockerfile's code to run a command by default when the container is executed

In this case, the instruction `docker run <image>` makes the container run the `ls` command. It is possible to overwrite what we pass to the entrypoint with `docker run <image> <cmd>`. Taking this into account, researchers could, for example, execute a certain script when the container is started.

Based on this, we have developed a script that installs OpenMPI and Horovod as follows:

1. Takes the versions of OpenMPI and Horovod to install provided by the user as arguments.
2. If they are already installed, skip the process, otherwise proceed with the installation.
3. Execute the command entered by the user as an argument (if none is entered, execute `/bin/bash`).

It is possible to include this script as entrypoint in the Dockerfile so when researchers run the container it installs the OpenMPI and Horovod versions requested by them, when the installation finishes the container tool (e.g., Docker, udocker) executes the command passed as `cmd` argument. This allows having an image without OpenMPI and Horovod installed, which gives a lot of flexibility since researchers can define their image without taking into account the OpenMPI version available in the cluster, making their image cluster-agnostic. When researchers start working in a specific cluster, they just need to run the container, taking into account the version of OpenMPI available there. The first time they run the container it installs the versions required by the cluster, later executions do not need to install them. The installation of Horovod is needed given its dependency on OpenMPI version.

In this section, we offered three alternatives to solve the dependency problem between OpenMPI versions in the image and the cluster. Tags are a simple solution, although it can complicate the researcher's workflow since they have to manage multiple images very similar to each other. The NGC catalog offers images ready to run this type of applications but limits the user to the software already installed in them. The entrypoint approach seems to be the best option given its flexibility, since the user ends up having a single image that

can be executed in any cluster regardless of the OpenMPI version installed in it.

4.3 DockerHub

In addition to offering tools for building images, Docker also offers DockerHub, a hosted repository service that enables their distribution. Through this, it is possible to upload images and share them with the rest of the world through its push and pull system. Here we can also find already built images that we can use as a basis to build our own.

Once researchers have developed the Dockerfile of their experiment, they can build the corresponding image and push it to DockerHub. When researchers work in an environment other than the local one, it is enough to pull the image from DockerHub to have access to their work environment. Within our developed workflow, this repository can be used as a download point for images. When researchers start working on an HPC cluster, they just need to pull the image from this repository and run the corresponding container to have a functional environment to train models in multiGPU settings.

DockerHub can also be useful for other researchers as they have direct access to the work environment of their colleagues, which facilitates both the replication of experiments and the development of new ones based on already developed computing environments.

In addition to DockerHub, with udocker researchers can use any other private repository to pull images from. In this work we focus on DockerHub, but the extension to a different repository is trivial.

4.4 Dockerfile Good Practices

The Dockerfile is composed of code so, as in this kind of project, it is important to follow best practices to produce clean, understandable and functional code. Docker includes in its official documentation the best practices for Dockerfile[5]. If these recommended practices are not followed, we will generate a poor quality Dockerfile. According to the nomenclature of configuration smells [40], the instructions that contribute to deteriorating this quality are known as Dockerfile smells.

In [41] these smells are classified into two main groups: DL-smells and SC-smells. The former is related to the violation of Docker best practices and contributes to building failures, increased build latency and security issues. The latter is related to best practices of shell scripting, a fundamental pillar of Dockerfile, its violation can cause strange behaviors and failures under certain conditions.

In [41] a study of the number of projects affected by these smells is carried out. It concludes that 84% of the Dockerfiles studied have some kind of smell. In the case of our developed workflow, it is important to control this mainly for two reasons. The first is to make our work environment understandable

---

[5]https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

and ease maintainability. The second is to generate a clean and light image, since in our workflow we transmit the image over the network (DockerHub) between our local environment and the HPC clusters.

Some specified smells contribute to increasing the size of the images, these are known as temporary file smells [42]. We have already seen that each command in the Dockerfile generates a layer that once stacked is read-only. Therefore, Docker images can be considered as append-only. If in one of these layers we generate a temporary file and subsequently we delete it in another, the space is not freed, only its reference is removed (the original layer with the temporary file is read-only and has its own filesystem). Therefore, if we want to delete temporary files, we must do it in the same layer in which they are generated, that is, the same Dockerfile command that generates those files is the one that must also delete them.

There are many ways to avoid these smells: using `ADD` instead of `COPY` when we want to decompress a local file in our image, avoiding the use of temporary files, or always ending the shell command with an `rm`. It is important to take all this into account to reduce the time elapsed on push and pulls.

Some tools help us detect and correct Dockerfile smells, one of these is Haskell Dockerfile Linter. This parses the Dockerfile into an abstract syntax tree and applies rules to see if best practices are followed, if not, it detects the smell and provides the solution to the user. There is an online version[6] in which the Dockerfile is pasted and returns the smells and their solutions. Researchers can use this tool to remove smells, especially those that contribute to large image size. This can be done once the Dockerfile has been developed, so the image pushed to Dockerhub is already the optimized one, preferably through software quality assurance tools [43].

4.5 udocker

As we have already seen, the dependence on administrator intervention or recent Linux kernel features of most of the containerization tools made udocker a perfect candidate for virtualization: it only depends on Python, can be executed under different execution modes and can be installed with not more than the user's privileges. Hereby, udocker is in charge of providing us with the *virtualized* runtime environment.

To work with udocker, researchers must install it on the cluster where they will launch their experiments. The main requirement is the availability of Python, which should be straightforward given that this programming language is currently the most popular according to the PopularitY of Programming Language (PYPL) index[7], especially in data science and scientific computing fields. It also requires some basic tools like `tar`, `find` and `curl` which are usually available in such environments. udocker can be installed in multiple ways such as GitHub, tarballs or pip. Figure 3 shows an example of installation via GitHub. This code downloads and executes the `udocker`

---

[6]https://hadolint.github.io/hadolint/
[7]https://pypl.github.io/PYPL.html

Python script and installs the required dependencies. Once this is done, researchers have an executable that can use to run all the udocker commands.

```
1    # Clone GitHub repository
2    git clone --depth=1 \
3    https://github.com/indigo-dc/udocker.git
4
5    # Create symbolic link
6    (cd udocker/udocker; ln -s maincmd.py udocker)
7
8    # Add to path
9    export PATH="pwd"/udocker/udocker:$PATH
0
1    # Install udocker dependencies
2    udocker install
```

Fig. 3 Bash script to install udocker via GitHub

To run a container, udocker uses a syntax similar to Docker. Suppose researchers have developed a Dockerfile under this workflow, they have uploaded the image to DockerHub and installed udocker on the cluster. To create a container following, for example, the approach of Section 4.2.3, they just need to execute the code of Fig.4. First, the image is pulled from Docker-Hub to the cluster and the corresponding container is created. Then with the command `udocker --setup nvidia` the necessary host libraries are added to the container so that it has access to the GPUs of the node. Finally, the container is executed, specifying the version of OpenMPI (the same as in the cluster) and Horovod to install. Once finished, a container ready to distribute the training of deep learning models in the cluster is configured.

To distribute the training, OpenMPI submits the processes as containers, so in each node a different copy of the container can be found. These containers can communicate between each other, allowing for parallel computation. Horovod coordinates distributed work via MPI, so in this case it is straightforward to communicate the different deployed containers as workers, allowing for the parallelization of the training. It is possible to use the `horovodrun` API or directly OpenMPI through `mpirun` and `mpiexec`. In the official Horovod documentation[8] researchers can find how to execute their container through these two approaches. This is straightforward and requires no adjustments. We recommend using the OpenMPI approach directly to have more control over the execution of the processes.

Figure 5 shows an example of code to run an experiment in an HPC cluster via udocker. In this case, the execution of a script is being distributed across 2 nodes with 2 GPUs each (4 GPUs in total). Given that this is executed through a job in a Slurm batch system, the `SBATCH` directives must be defined. These depend on the cluster configuration and the batch system in use, although

---

[8]https://horovod.readthedocs.io/en/stable/

```
1    # Pull the image
2    udocker pull <image>
3
4    # Create the container
5    udocker create \
6    --name=<container_name> <image>
7
8    # Add the GPU libraries to the container
9    udocker setup --nvidia --force \
0    <container_name>
1
2    # Install OpenMPI and Horovod in
3    # the container
4    udocker run \
5    --env="OpenMPI=<HPC_version>" \
6    --env="HOROVOD=latest" \
7    <container_name>
```

Fig. 4 Bash script to create and configure an udocker container

they do not usually vary much. Then the cluster's OpenMPI is loaded and the `mpirun` parameters defined (we took as reference the Horovod documentation mentioned before). By default, all communication between GPUs is carried out through NCCL and the rest through MPI, taking advantage of low latency communications through specialized interconnects such as Infiniband.

Finally, `udocker run` is executed through `mpirun` which is responsible for distributing the container in the selected nodes and communicating all the processes. udocker takes as input the information related to the researcher's user in the cluster, the path to the volume to mount in the container and the command to execute. If this command runs a Python script with a Horovod adapted deep learning training, it will be automatically distributed across the selected nodes.

The conjunction between DockerHub and udocker provides researchers with an easily shareable workflow between their local and HPC environment in which they will run their experiments. They can develop their image locally with Docker, upload it to DockerHub and access it through udocker on any HPC cluster. They could also run their experiment with Docker on a small scale in their local environment to verify that everything works correctly and once it is ready, upload it to DockerHub to pull it to the HPC cluster and run it. Once researchers own a ready-to-use image of their working environment, they can work across multiple clusters without the need for specific installations or root access, they just need to pull the image from DockerHub to the respective cluster and start working with their custom environment.

This methodology provides researchers with a dynamic workflow between local environments and HPC clusters, avoiding complex installations based on the environment used (which would be the case if containerization is not used)

```
1    #!/bin/bash
2    #
3    #SBATCH --job-name=train-model
4    #SBATCH --partition=wngpu
5    #SBATCH --gres=gpu:2
6    #SBATCH --ntasks=4
7    #SBATCH --tasks-per-node=2
8    #SBATCH --output=train-model.out
9    #SBATCH --error=train-model.err
0
1    # Load cluster OpenMPI
2    module load OPENMPI/4.1.0
3
4    # mpirun configuration
5    MPI_PARAMS="-bind-to none -map-by slot \
6    -x LD_LIBRARY_PATH -x PATH \
7    -x HOROVOD_MPI_THREADS_DISABLE=1 \
8    -mca pmix_server_usock_connections 1 \
9    -mca pml ob1 -mca btl ^openib"
0
1    # Run the experiment
2    mpirun $MPI_PARAMS \
3    udocker run \
4    --hostenv --hostauth --user=$USER \
5    --volume=$HOME/experiment/:/home/ \
6    multigpu-base \
7    python /home/train_model.py
```

Fig. 5 Example of bash code to run an experiment through an udocker container in an HPC cluster using Slurm

and providing them with a closed environment that facilitates the deployment of experiments in a cluster-agnostic way.

5 Evaluation

In this section we present the developed experiments to assess the characteristics of the workflow presented in this work: cluster-agnosticism and scaling efficiency. For the former, different scientific workflows are executed in different clusters, using the same starting image as well as the same code. For the latter, different deep learning models are trained in various multiGPU distributed environments with the aim to study its scalability. The code[9] (including Dockerfiles) and the Docker images[10] for these experiments are publicly available.

---

[9] https://github.com/IFCA/workflow-DL-HPC
[10] https://hub.docker.com/r/gonzabad/multigpu-horovod

5.1 TensorFlow Benchmark

Horovod includes TensorFlow Benchmark[11], an implementation of SOTA deep learning models used to test training performance both in CPU and GPU environments. This benchmark allows training models like ResNet [44], InceptionV3 [45] and DenseNet [46] on synthetic and real data. Particularly, we have trained ResNet50, ResNet101 and InceptionV3 on synthetic data to avoid disk I/O uncertainty. Table 1 shows key characteristics of the models employed.

Model	Number of Parameters	Batch size per GPU
InceptionV3	23,851,784	256
ResNet50	25,636,712	256
ResNet101	44,707,176	128

Table 1 Characteristics of the models trained in the TensorFlow Benchmark experiment

The objective of this experiment is to verify that the containerization environment scales correctly with respect to the number of GPUs and the number of nodes in comparison with a non-containerized scenario. For this, we train these models in a containerized and non-containerized (native) environments. The former is implemented through the developed workflow, while the latter is not.

This experiment requires a basic environment with a GPU based Tensorflow and Horovod installation. For the workflow test we have prepared a basic image[12] satisfying these requisites (OpenMPI and Horovod versions issue is addressed by the technique described in Section 4.2.3). This image can be used as a basis for building more complex environments. udocker allows the use of different execution modes for running the containers, we have opted for the default (P1) which uses the PRoot model with SECCOMP filtering (we refer the reader to [14] for more details). For the native environment we had to depend on software already installed on the HPC cluster, in addition to manually performed installations. For this last part we relied on Anaconda[13], one of the most popular set of packages and tools among the data science community. This installation is generally more complex than our proposed method, since we heavily depend on the versions of the software installed in the cluster. For example, the version of cudatoolkit available through Anaconda does not include the NVIDIA CUDA compiler (NVCC), so users must install it manually from the NVIDIA repositories, taking into account the actual driver and CUDA version of the cluster. Dependency management is also a complex task, requiring specific configurations to avoid incompatibilities between the different components of the NVIDIA toolkit, as well as requiring the configuration of additional libraries for MPI implementation. Horovod provides documentation for the proper configuration in a local environment[14].

---

[11]https://github.com/horovod/horovod/tree/master/examples/tensorflow2
[12]https://hub.docker.com/layers/gonzabad/multigpu-horovod/base/images/
sha256-2b04fcdb1f7727abb4ba67fcaa12aefe36d2606f1a25ef97bcab8be6ae7e7c00?context=explore
[13]https://www.anaconda.com/
[14]https://horovod.readthedocs.io/en/stable/conda_include.html

5.2 Empirical Statistical Downscaling

Climate modelling plays a critical role in decision-making, in areas ranging from agriculture and health to tourism and economy. This information is generated by Global Circulation Models (GCMs), mathematical models that compute the climate system components and their interactions. These models are highly complex, so the generated predictions suffer from a coarse spatial resolution. Sometimes it is necessary for this information to be available at a higher-resolution in order to make decisions on local scale. Empirical statistical downscaling (ESD, [47]) methods aim to learn a statistical relationship between a set of low-resolution variables and a high-resolution variable of interest. When these variables lean on observational records, we work under the *perfect-prognosis* approach (PP-ESD).

Given the success of deep learning in fields like computer vision, these kinds of techniques have recently been successfully applied to PP-ESD [48–50]. In particular, convolutional neural networks have been positioned as a promising approach given the ability to model relationships where the data present a spatial structure.

We develop a PP-ESD experiment following the framework presented in [49]. We downscale precipitation over the region of North America using deep learning techniques. As predictor we use five thermodynamical variables at four different vertical levels of the ERA-Interim reanalysis dataset [51] (2° resolution) and as predictand the daily accumulated precipitation from EWEMBI [52] (0.5° resolution). We train the model in the period spanned by the years 1979-2002. ERA-Interim data can be downloaded from the ECMWF website[15] and EWEMBI dataset is available at ISIMIP[16].

We have developed a custom model exclusively composed of convolutional layers, what makes it fully-convolutional. Our objective with this experiment is not to improve the accuracy of previous models, but to prove that the workflow possess benefits for a real scientific case. Table 2 shows details of the model trained.

Number of Parameters	Input Size	Output Size	Batch Size per GPU	Optimizer	Learning Rate
1,615,671	(30, 53, 20)	(117, 209, 3)	32	Adam	0.0001

Table 2 Characteristics of the fully-convolutional model trained in the Statistical Downscaling experiment

In this case we work with R and Python. The former is used to pre-process and post-process the data, while the latter is responsible to distribute the training of the model (via Horovod). Training has been implemented with Keras, a high level API of TensorFlow. To adapt it to Horovod we only needed to add a few lines of code. This has been done following the official Horovod documentation. In this experiment the use of R is also justified by climate4R [53], a

---

[15]https://www.ecmwf.int/
[16]https://www.isimip.org/

bundle of R packages for access, pre-process, post-process and visualization of climate data. In this case, to configure the environment we just need to add to the Dockerfile the commands required to install R and climate4R. As a result, we get a ready-to-use image satisfying the requirements of our experiment.

The goal of this experiment is to show how the workflow can help in moving experiments between clusters while maintaining its scalability in multiGPU environments. This experiment was executed in a different cluster from the previous one, however it was not necessary to apply substantial changes to the workflow. The image built for this experiment[17] was developed using as basis the image from the TensorFlow Benchmark experiment, solely adding the software required for this second experiment. The scheduling of the jobs has been done in the same way in both experiments (following the indications of section 4.5), modifying only the SBATCH directives since they depend on specific configurations of each cluster. This has been possible thanks to the workflow developed and its cluster-agnosticism feature.

6 Cluster Setup

The experiments have been executed on two different computing clusters: one at the Spanish National Research Council (CSIC) deep learning computing infrastructure located at the Institute of Physics of Cantabria (IFCA, CSIC–UC) in Spain and ForHLR2 at the Karlsruhe Institute of Technology (KIT) in Germany. Within these clusters the nodes with GPUs are the ones accessed. Table 3 shows the specifications of each of these two clusters.

	CSIC DL Infrastructure	**ForHLR2**
CPU	Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GH	Intel Xeon E7-4830 v3 processors @ 2.10GH
Memory	64 GB	1 TB
GPU	NVIDIA Tesla V100	NVIDIA GeForce GTX980 Ti
GPU Memory	32 GB	6 GB
GPUs per node	2	4
GPU nodes	20	21
Connection	InfiniBand EDR NVIDIA Mellanox ConnectX-5 cards	InfiniBand 4X EDR Interconnect
Operating system	Ubuntu 18.04.4 LTS	Red Hat Enterprise Linux (RHEL) 7.x
Storage	General Parallel File System (GPFS [54])	Lustre [55]

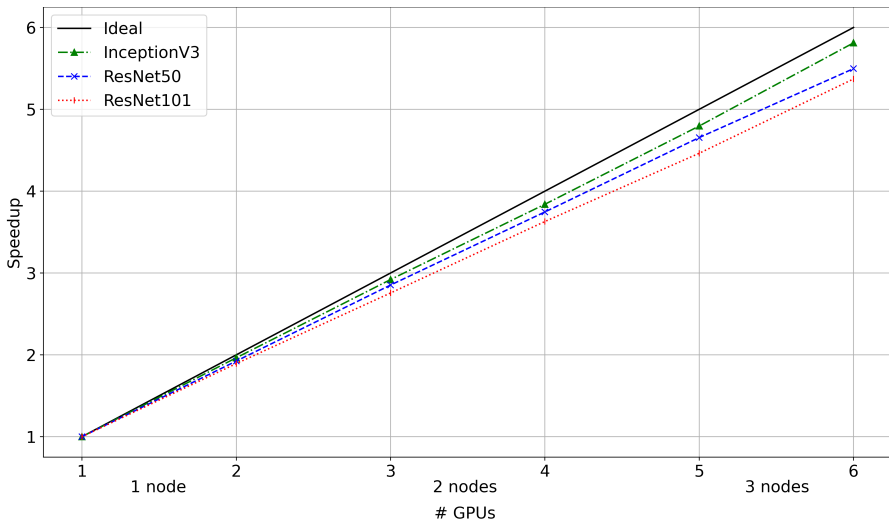**Table 3** Specifications (rows) for the two different clusters (columns) accessed for the different experiments

The experiments of Section 5.1 have been executed at IFCA while those of Section 5.2 at ForHLR2.

---

[17]https://hub.docker.com/layers/gonzabad/multigpu-horovod/downscaling/images/sha256-d695141efc6677e0ac38e6702c5a0c580ed8bc7613a6e5063e609c3e83a738b1?context=explore

7 Results

7.1 TensorFlow Benchmark

For this experiment, we have trained the three models mentioned in Section 5.1 in configurations of 1, 2, 3, 4, 5 and 6 GPUs. Training sessions consisted of 10 warm-up iterations (not counting towards measurements) and 10 iterations from which the final measures were obtained. Each iteration processed 10 batches of data, each with the number of synthetic samples specified in Table 1.
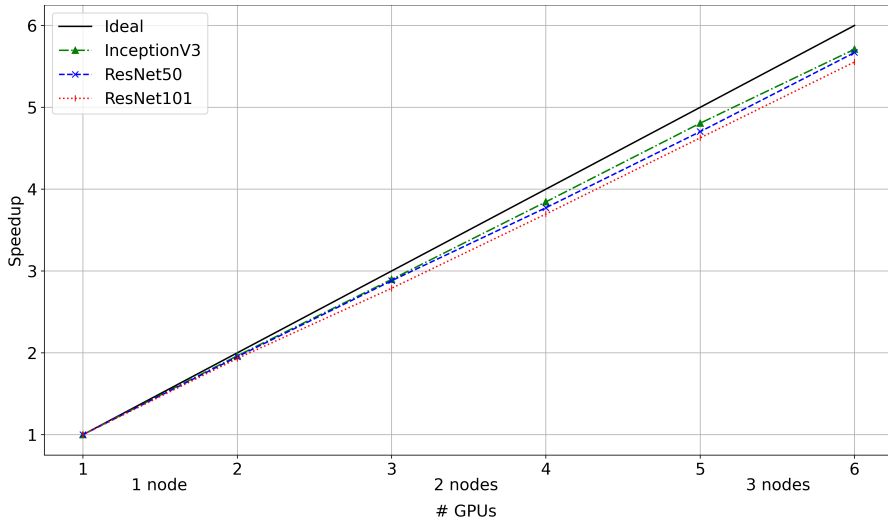

Fig. 6 Speedup plot for the TensorFlow Benchmark experiment using udocker over the different GPU configurations

Figure 6 shows the speedup achieved for the six configurations concerning the number of images processed per second when training the models under the udocker workflow. For one and two nodes the speedup in training is very close to the ideal. For the configurations running on three nodes, although not ideal, acceptable results are achieved. Among the models, ResNet101 is the one that scales the worst, followed by the ResNet50 and InceptionV3. This order coincides with the number of parameters, the more parameters the model has, the lower the speedup.

Figure 7 shows the results for the same experiment but in the native environment. These depict a similar trend in the scalability with respect to the workflow experiment, however the speedup seems to be a little bit closer to the ideal one. In addition, the difference in speedup between the three models is smaller, although it can still be appreciated.

Finally, Figures 8, 9 and 10 compare the images processed per second in the different configurations under the udocker workflow and the native

Fig. 7 Speedup plot for the TensorFlow Benchmark experiment using the native environment over the different GPU configurations

environment. Values represented as bars correspond to the mean of images per second across the different iterations. Within each bar it can be seen the 95% confidence interval for these measurements. Concerning udocker and native environments, no significant differences are observed between models, especially for one and two nodes. For three nodes a maximum difference of 3% of processed images in favor of the native environment is observed, although differences benefiting the udocker workflow can also be found (e.g., InceptionV3).

7.2 Statistical Downscaling

This experiment has been executed in the ForHLR2 cluster. The GPUs available had 6 GB of memory, which was not enough to train the model for the statistical downscaling task when using a batch size of more than 32 samples. We have avoided this issue by scaling the training to more than one GPU. More specifically, we ran the experiment in configurations of 1, 2, 3 and 4 GPUs accessing a single node. For this, we have deployed the distributed training across GPUs with the udocker workflow. Although all GPUs were located on a single node, we still benefited from the Horovod framework. This has allowed us to work with batch sizes of 32, 64, 96 and 128 samples (1, 2, 3 and 4 GPUs respectively).
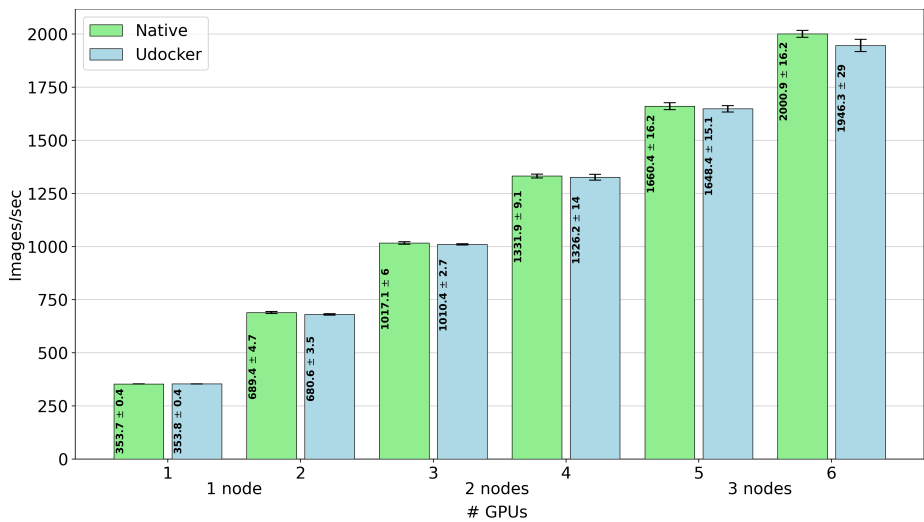
Figure 11 shows the number of images processed per second concerning the number of GPUs. Results depict an optimal scalability given the similarity between the ideal scenario and the amount of images processed.

Figure 12 shows the evolution of the loss during training with respect to the number of GPUs. For 1, 2 and 3 GPUs (effective batch size of 32, 64 and 96 respectively) a similar convergence rate can be seen, however for 4 GPUs a
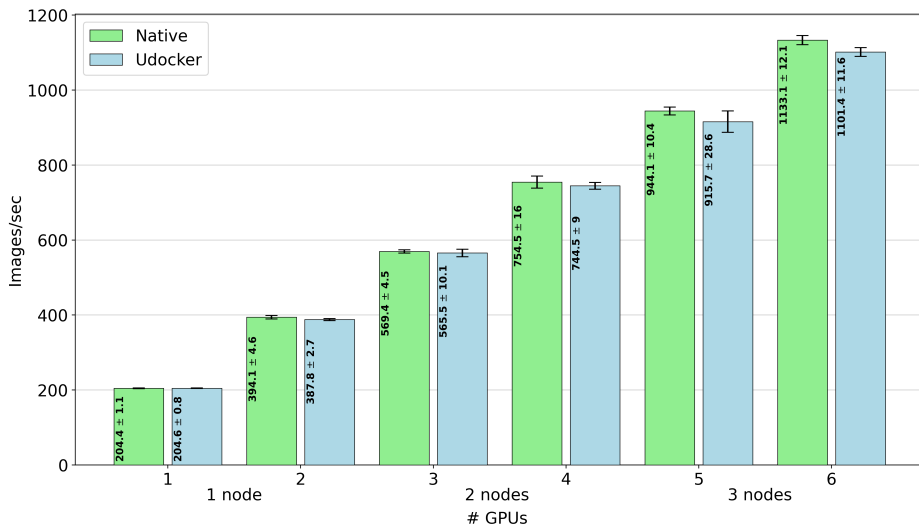
Fig. 8 Comparison of images processed per second when training the InceptionV3 model. The values represented correspond to the mean across different iterations. The 95% confidence interval can be found within each bar.
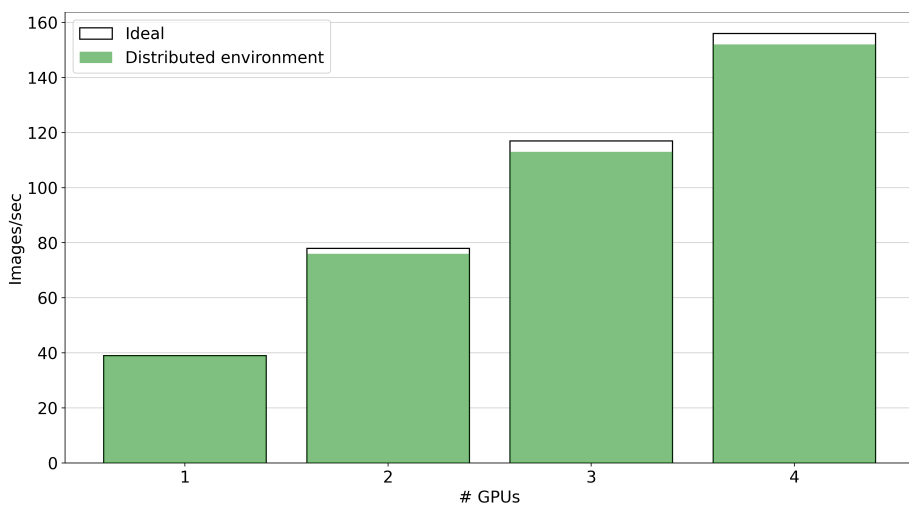

Fig. 9 Comparison of images processed per second when training the ResNet50 model. The values represented correspond to the mean across different iterations. The 95% confidence interval can be found within each bar.

slower evolution of the loss is observed, although it ends up reaching the same value.
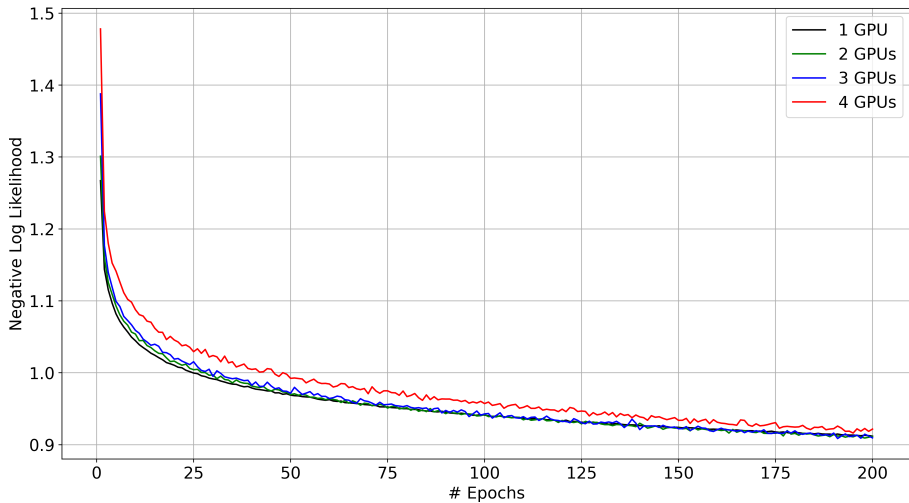
Fig. 10 Comparison of images processed per second when training the ResNet101 model. The values represented correspond to the mean across different iterations. The 95% confidence interval can be found within each bar.

Fig. 11 Images processed per second when training the statistical downscaling model using udocker

8 Discussion

The presented results show that it is possible to efficiently distribute the training of deep learning models in HPC clusters with the developed workflow. The differences observed in performance between udocker and the native environment are minimal, highlighting an advantage of less than 3% in images processed per second towards the latter. This is an expected effect, since any

Fig. 12 Training Loss (Negative Log Likelihood) evolution during training of the statistical downscaling model

type of containerization or virtualization can introduce overhead, although it is usually minimal [29]. No comparisons have been made with models trained with other tools like Singularity, as it has been already seen that their performance is very similar [34].

Regarding the number of GPUs, the results show a small deviation from the ideal scalability. This is expected both in a containerized environment and in a native one since the distribution of the training introduces difficulties (i.e. communication between workers) that slightly damage the speedup. A worse scalability is also observed with respect to the number of parameters of the model, this is due to the increase in the communication time between workers: the greater the model, the more data (parameters) need to communicate over the network. This effect is most noticeable when using udocker, as additional operations are required between workers given the containerization layer.

Although in the case of statistical downscaling the different models achieve a similar loss value, they do it at different speeds. This is an effect generally caused by the use of large batch sizes [56]. Although advanced techniques exist to mitigate this effect, we followed the Horovod recommendations and used a linear scaling of the learning rate based on the number of GPUs, obtaining an acceptable learning result regardless of the batch size.

The statistical downscaling experiment has been executed in a different cluster than the TensorFlow benchmark, however both the deep learning framework and all the libraries it depends on have been installed following the same image. Also, the scripts to run the jobs have been shared between experiments. Despite significant differences between clusters, such as the GPU model or the operating system on which they run (see Section 6), there have been no compatibility issues and the experiments have been executed directly, proving the cluster-agnosticism of the workflow.

Installation and configuration of the required software in the clusters for both experiments has been carried out with no need of administrators intervention given the non-root nature of the tools conforming the workflow. The images of both experiments have been developed in a local machine, transferring them to the HPC cluster via Dockerhub. These facts enable users with the autonomy to control their experiments in a more dynamic way, without depending on external individuals.

9 Conclusions

This work presents a methodology and workflow based on containers to perform distributed training of deep learning models in HPC clusters. It covers the entire process required for this purpose, from the development of the environment to its distribution and execution. In this way, domain scientists who need to apply deep learning algorithms can take advantage of the computing power available in HPC clusters. We have shown that this method scales well with respect to the number of GPUs and nodes thanks to the use of udocker. We have conducted experiments with a real scientific use case proving that learning is not affected.

Using Docker containers to encapsulate the execution of scientific user applications poses benefits in order to port and execute the workloads in HPC environments. The workflow developed also offers a number of advantages over more conventional ones: improves transferability between different clusters, promotes scientific collaboration and sharing between communities, allows for a more dynamic research cycle given its non-root nature and permits the use of the latest software in a secure way at HPC clusters. With this work we hope to encourage domain scientists to make use of these computing resources in order to further advance their research.

Acknowledgments

References

[1] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2006)

[2] Baldi, P., Sadowski, P., Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning. Nature communications **5**(1), 1–9 (2014). https://doi.org/10.1038/ncomms5308

[3] de Oliveira, L., Kagan, M., Mackey, L., Nachman, B., Schwartzman, A.: Jet-images—deep learning edition. Journal of High Energy Physics **2016**(7), 1–32 (2016). https://doi.org/10.1007/JHEP07(2016)069

[4] Tuccillo, D., Huertas-Company, M., Decencière, E., Velasco-Forero, S., Domínguez Sánchez, H., Dimauro, P.: Deep learning for galaxy surface brightness profile fitting. Monthly Notices of the Royal Astronomical Society **475**(1), 894–909 (2018). https://doi.org/10.1093/mnras/stx3186

[5] Primack, J., Dekel, A., Koo, D., Lapiner, S., Ceverino, D., Simons, R., Snyder, G., Bernardi, M., Chen, Z., Domínguez-Sánchez, H., *et al.*: Deep learning identifies high-z galaxies in a central blue nugget phase in a characteristic mass range. The Astrophysical Journal **858**(2), 114 (2018)

[6] Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., *et al.*: Highly accurate protein structure prediction with alphafold. Nature **596**(7873), 583–589 (2021). https://doi.org/10.1038/s41586-021-03819-2

[7] Scher, S.: Toward data-driven weather and climate forecasting: Approximating a simple general circulation model with deep learning. Geophysical Research Letters **45**(22), 12–616 (2018). https://doi.org/10.1029/2018GL080704

[8] Rasp, S., Pritchard, M.S., Gentine, P.: Deep learning to represent subgrid processes in climate models. Proceedings of the National Academy of Sciences **115**(39), 9684–9689 (2018). https://doi.org/10.1073/pnas.1810286115

[9] López García, Á., Marco de Lucas, J., Antonacci, M., Zu Castell, W., David, M., Hardt, M., Lloret Iglesias, L., Moltó, G., Plociennik, M., Tran, V., Alic, A.S., Caballer, M., Campos Plasencia, I., Costantini, A., Dlugolinsky, S., Duma, D.C., Donvito, G., Gomes, J., Heredia Cacha, I., Ito, K., Kozlov, V.Y., Nguyen, G., Orviz Fernández, P., Šustr, Z., Wolniewicz, P.: A Cloud-Based Framework for Machine Learning Workloads and Applications. IEEE Access **8**, 18681–18692 (2020). https://doi.org/10.1109/ACCESS.2020.2964386

[10] Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux journal, 5 (2014)

[11] Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: Scientific containers for mobility of compute. PloS one **12**(5), 0177459 (2017). https://doi.org/10.1371/journal.pone.0177459

[12] Priedhorsky, R., Randles, T.: Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In: Proceedings of the International

Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–10 (2017). https://doi.org/10.1145/3126908.3126925

[13] Gerhardt, L., Bhimji, W., Canon, S., Fasel, M., Jacobsen, D., Mustafa, M., Porter, J., Tsulaia, V.: Shifter: Containers for hpc. In: Journal of Physics: Conference Series, vol. 898, p. 082021 (2017). https://doi.org/10.1088/1742-6596/898/8/082021. IOP Publishing

[14] Gomes, J., Bagnaschi, E., Campos, I., David, M., Alves, L., Martins, J., Pina, J., Lopez-Garcia, A., Orviz, P.: Enabling rootless linux containers in multi-user environments: the udocker tool. Computer Physics Communications **232**, 84–97 (2018). https://doi.org/10.1016/j.cpc.2018.05.021

[15] Younge, A.J., Pedretti, K., Grant, R.E., Brightwell, R.: A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 74–81 (2017). https://doi.org/10.1109/CloudCom.2017.40. IEEE

[16] Brayford, D., Vallecorsa, S., Atanasov, A., Baruffa, F., Riviera, W.: Deploying AI Frameworks on Secure HPC Systems with Containers. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6 (2019). https://doi.org/10.1109/HPEC.2019.8916576

[17] Brayford, D., Vallecorsa, S.: Deploying Scientific Al Networks at Petaflop Scale on Secure Large Scale HPC Production Systems with Containers. In: Proceedings of the Platform for Advanced Scientific Computing Conference. PASC '20, pp. 1–8. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3394277.3401850

[18] Sergeev, A., Del Balso, M.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799 [cs, stat] (2018). https://doi.org/10.48550/arXiv.1802.05799

[19] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 30 (2017)

[20] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., Aila, T.: Analyzing and Improving the Image Quality of StyleGAN. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 8107–8116 (2020). https://doi.org/10.1109/CVPR42600.2020.00813

[21] Nguyen, G., Dlugolinsky, S., Bobák, M., Tran, V., López García, Á., Heredia, I., Malík, P., Hluchý, L.: Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. Artificial Intelligence Review **52**(1), 77–124 (2019). https://doi.org/10.1007/s10462-018-09679-z

[22] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283. USENIX Association, Savannah, GA (2016)

[23] Chollet, F., et al.: Keras. https://keras.io (2015)

[24] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach, H., Larochelle, H., Beygelzimer, A., Alché-Buc, F.d., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32 (2019)

[25] Patarasuk, P., Yuan, X.: Bandwidth optimal all-reduce algorithms for clusters of workstations. Journal of Parallel and Distributed Computing **69**(2), 117–124 (2009). https://doi.org/10.1016/j.jpdc.2008.09.002

[26] Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, pp. 44–60. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/10968987_3

[27] Oesterle, F., Ostermann, S., Prodan, R., Mayr, G.J.: Experiences with distributed computing for meteorological applications: grid computing and cloud computing. Geoscientific Model Development **8**(7), 2067–2078 (2015). https://doi.org/10.5194/gmd-8-2067-2015

[28] Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172 (2015). https://doi.org/10.1109/ISPASS.2015.7095802

[29] Torrez, A., Randles, T., Priedhorsky, R.: HPC Container Runtimes have Minimal or No Performance Impact. In: 2019 IEEE/ACM International

Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp. 37–42 (2019). https://doi.org/10.1109/CANOPIE-HPC49598.2019.00010

[30] Chung, M.T., Quang-Hung, N., Nguyen, M.-T., Thoai, N.: Using Docker in high performance computing applications. In: 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), pp. 52–57 (2016). https://doi.org/10.1109/CCE.2016.7562612

[31] Sparks, J.: Enabling Docker for HPC. Concurrency and Computation: Practice and Experience **31**(16), 5018 (2019). https://doi.org/10.1002/cpe.5018

[32] Azab, A.: Enabling Docker Containers for High-Performance and Many-Task Computing. In: 2017 IEEE International Conference on Cloud Engineering (IC2E), pp. 279–285 (2017). https://doi.org/10.1109/IC2E.2017.52

[33] Jacobsen, D.M., Canon, R.S.: Contain This, Unleashing Docker for HPC. In: Proceedings of the Cray User Group, p. 8 (2015)

[34] Grupp, A., Kozlov, V., Campos, I., David, M., Gomes, J., López García, Á.: Benchmarking Deep Learning Infrastructures by Means of Tensor-Flow and Containers. In: Weiland, M., Juckeland, G., Alam, S., Jagode, H. (eds.) High Performance Computing. Lecture Notes in Computer Science, pp. 478–489. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34356-9_36

[35] Gantikow, H., Walter, S., Reich, C.: Rootless containers with podman for hpc. In: Springer (ed.) International Conference on High Performance Computing, pp. 343–354 (2020). https://doi.org/10.1007/978-3-030-59851-8_23

[36] Gamblin, T., LeGendre, M., Collette, M.R., Lee, G.L., Moody, A., De Supinski, B.R., Futral, S.: The spack package manager: bringing order to hpc software chaos. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2015). https://doi.org/10.1145/2807591.2807623

[37] Höb, M., Kranzlmüller, D.: Enabling easey deployment of containerized applications for future hpc systems. In: Springer (ed.) International Conference on Computational Science, pp. 206–219 (2020). https://doi.org/10.1007/978-3-030-50371-0_15

[38] Canon, R.S., Younge, A.: A case for portability and reproducibility of hpc containers. In: IEEE (ed.) 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments

in HPC (CANOPIE-HPC), pp. 49–54 (2019). https://doi.org/10.1109/CANOPIE-HPC49598.2019.00012

[39] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (2004). https://doi.org/10.1007/978-3-540-30218-6_19

[40] Sharma, T., Fragkoulis, M., Spinellis, D.: Does Your Configuration Code Smell? In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 189–200 (2016)

[41] Wu, Y., Zhang, Y., Wang, T., Wang, H.: Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study. IEEE Access **8**, 34127–34139 (2020). https://doi.org/10.1109/ACCESS.2020.2973750

[42] Lu, Z., Xu, J., Wu, Y., Wang, T., Huang, T.: An Empirical Case Study on the Temporary File Smell in Dockerfiles. IEEE Access **7**, 63650–63659 (2019). https://doi.org/10.1109/ACCESS.2019.2905424

[43] Orviz Fernández, P., David, M., Duma, D.C., Ronchieri, E., Gomes, J., Salomoni, D.: Software Quality Assurance in INDIGO-DataCloud Project: a Converging Evolution of Software Engineering Practices to Support European Research e-Infrastructures. Journal of Grid Computing **18**(1), 81–98 (2020). https://doi.org/10.1007/s10723-020-09509-z

[44] He, K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90

[45] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1–9 (2015). https://doi.org/10.1109/CVPR.2015.7298594

[46] Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely Connected Convolutional Networks. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2261–2269 (2017). https://doi.org/10.1109/CVPR.2017.243

[47] Maraun, D., Widmann, M.: Statistical Downscaling and Bias Correction for Climate Research. Cambridge University Press, (2018)

[48] Vandal, T., Kodra, E., Ganguly, S., Michaelis, A., Nemani, R., Ganguly, A.R.: DeepSD: Generating High Resolution Climate Change Projections through Single Image Super-Resolution. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery And Data Mining. KDD '17, pp. 1663–1672. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3097983.3098004

[49] Baño-Medina, J., Manzanas, R., Gutiérrez, J.M.: Configuration and intercomparison of deep learning neural models for statistical downscaling. Geoscientific Model Development **13**(4), 2109–2124 (2020). https://doi.org/10.5194/gmd-13-2109-2020

[50] Sun, L., Lan, Y.: Statistical downscaling of daily temperature and precipitation over China using deep learning neural models: Localization and comparison with other methods. International Journal of Climatology **41**(2), 1128–1147 (2021). https://doi.org/10.1002/joc.6769

[51] Dee, D.P., Uppala, S.M., Simmons, A.J., Berrisford, P., Poli, P., Kobayashi, S., Andrae, U., Balmaseda, M.A., Balsamo, G., Bauer, P., Bechtold, P., Beljaars, A.C.M., Berg, L.v.d., Bidlot, J., Bormann, N., Delsol, C., Dragani, R., Fuentes, M., Geer, A.J., Haimberger, L., Healy, S.B., Hersbach, H., Hólm, E.V., Isaksen, L., Kållberg, P., Köhler, M., Matricardi, M., McNally, A.P., Monge-Sanz, B.M., Morcrette, J.-J., Park, B.-K., Peubey, C., Rosnay, P.d., Tavolato, C., Thépaut, J.-N., Vitart, F.: The ERA-Interim reanalysis: configuration and performance of the data assimilation system. Quarterly Journal of the Royal Meteorological Society **137**(656), 553–597 (2011). https://doi.org/10.1002/qj.828

[52] Lange, S.: EartH2Observe, WFDEI and ERA-Interim data Merged and Bias-corrected for ISIMIP (EWEMBI). GFZ Data Services (2019). https://doi.org/10.5880/PIK.2019.004

[53] Iturbide, M., Bedia, J., Herrera, S., Baño-Medina, J., Fernández, J., Frías, M.D., Manzanas, R., San-Martín, D., Cimadevilla, E., Cofiño, A.S., Gutiérrez, J.M.: The R-based climate4R open framework for reproducible climate data access and post-processing. Environmental Modelling & Software **111**, 42–54 (2019). https://doi.org/10.1016/j.envsoft.2018.09.009

[54] Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: Conference on File and Storage Technologies (FAST 02) (2002)

[55] Braam, P.: The lustre storage architecture. arXiv preprint arXiv:1903.01955 (2019). https://doi.org/10.48550/arXiv.1903.01955

[56] Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P.:

On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836 (2016). https://doi.org/10.48550/arXiv.1609.04836