2021-02

Container orchestration on HPC systems through Kubernetes

Zhou Naweiluo, Georgiou Yiannis, Pospieszny Marcin, Zhong Li, Zhou Huan, Niethammer Christoph, Pejak Branislav, Marko Oskar, Hoppe Dennis

Springer

Zhou, Naweiluo, Georgiou, Yiannis, Pospieszny, Marcin, Zhong, Li, Zhou, Huan, et al. 2021. Container orchestration on HPC systems through Kubernetes. Journal of Cloud Computing 10(1): 1–14. doi: 10.1186/s13677-021-00231-z.
https://open.uns.ac.rs/handle/123456789/32693

Zhou *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2021) 10:16
https://doi.org/10.1186/s13677-021-00231-z

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

Container orchestration on HPC systems through Kubernetes

Naweiluo Zhou[1*], Yiannis Georgiou[2], Marcin Pospieszny[3], Li Zhong[1], Huan Zhou[1], Christoph Niethammer[1], Branislav Pejak[4], Oskar Marko[4] and Dennis Hoppe[1]

Abstract

Containerisation demonstrates its efficiency in application deployment in Cloud Computing. Containers can encapsulate complex programs with their dependencies in isolated environments making applications more portable, hence are being adopted in High Performance Computing (HPC) clusters. *Singularity*, initially designed for HPC systems, has become their *de facto* standard container runtime. Nevertheless, conventional HPC workload managers lack micro-service support and deeply-integrated container management, as opposed to container orchestrators. We introduce a Torque-Operator which serves as a bridge between HPC workload manager (TORQUE) and container orchestrator (Kubernetes). We propose a hybrid architecture that integrates HPC and Cloud clusters seamlessly with little interference to HPC systems where container orchestration is performed on two levels.

Keywords: Cloud computing, HPC workload manager, Container orchestration, TORQUE, Kubernetes, Singularity

Introduction

Cloud computing demands high-portability. Containerisation ensures environment compatibility by encapsulating applications together with their libraries, configuration files and other dependencies [1], thus enabling users to move and deploy programs easily among clusters. Containerisation is a virtualisation technology [2]. Rather than simulating the holistic operating system (OS) as in a Virtual Machine (VM), containers only share the host OS. This feature makes containers more lightweight than VMs. Containers are dedicated to run *micro-services* [3] and one container mostly hosts one application. Nevertheless, containerised applications can become complex, e.g. thousands of separate containers may be required in production. The production can benefit from container orchestrators that can provide efficient environment provisioning and auto-scaling [4].

Big Data Analytics hosted on Cloud are compute-intensive or data-intensive, mainly due to deployment of Artificial Intelligence (AI) or Machine Learning (ML) applications, which demand extremely fast knowledge extraction in order to make rapid and accurate decisions. High Performance Computing (HPC) systems are traditionally applied to perform large-scale financial and engineering simulation that demand low latency and high throughput. HPC is not ready to fully support AI applications due to their complex environment requirements. In the muti-tenant environment of an HPC cluster, it is difficult to install new software packages since this may alter working environments of existing users and even raise security concerns. Furthermore, HPC systems, especially HPC production systems, usually provide a complete stack of software packages which often do not allow user customisation. Containerising AI applications can be a potential solution.

Nevertheless, typical HPC jobs are large workloads that are normally hardware-specific. HPC jobs are often submitted to a batch queue within a workload manager where jobs wait to be scheduled from minutes to days. An HPC cluster is typically equipped with a workload manager. A *workload manager* is composed of a *resource manager* and a *job scheduler*. A resource manager [5] allocates resources

*Correspondence: naweiluo.zhou@hlrs.de
[1]High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Stuttgart, Germany
Full list of author information is available at the end of the article

(e.g. CPU and memory), schedules jobs and guarantees no interference from other user processes. A job scheduler determines the job priorities, enforces resource limits and dispatches jobs to available nodes [6]. A container orchestrator, such as Kubernetes [3], on its own does not address all the requirements of HPC systems, therefore, cannot replace existing workload managers in HPC centres. HPC workload managers, such as TORQUE, lack micro-service support and deeply-integrated container management capabilities in which container orchestrators manifest their efficiency.

This work contributes to the following aspects.

- We present a hybrid architecture that is composed of an HPC cluster and a Cloud cluster, where container orchestration on the HPC cluster can be performed by the container orchestrator (i.e. Kubernetes) located in the Cloud cluster. Little modification is required on the HPC systems;
- We propose a dual-level scheduling for container jobs on HPC systems;
- We describe the implementation of a tool named *Torque-Operator* which bridges TORQUE and Kubernetes.

The rest of the article is organised as follows. First, we present the background on the key technologies and the challenges in "Background" section. Next, "Related work" section briefly reviews the work of state of the art. We describe the proposed architecture and the structure of Torque-Operator in "Architecture and tool description" section. Following that, performance evaluation is given in "Use cases" section. Last, "Conclusion and future work" section concludes this paper and proposes future work.

Background

This section presents a concise background of technologies and techniques on orchestration of HPC and Cloud clusters, which leads to the motivation description of connecting Cloud clusters with HPC clusters. Orchestration under this context means automated configuration, coordination and management of HPC systems and Cloud computing systems.

Workload managers for HPC

A key component of an HPC system is its workload manager. Slurm [7] and TORQUE [8] are the two main-stream workload managers. Often TORQUE is coupled with a complex job scheduler, e.g. Moab [9].

TORQUE

The structure of a TORQUE managed cluster consists of a head node and many compute nodes as illustrated in Fig. 1 where only three computer nodes are shown. The

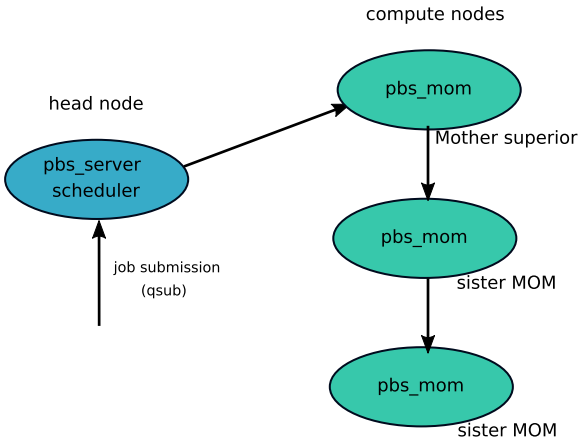


Fig. 1 TORQUE structure. *pbs_server*, scheduler and *pbs_mom* are the daemons running on the nodes. Mother Superior is the first node on the node list that finds by the scheduler

head node (coloured in blue in Fig. 1) controls the entire TORQUE system. A *pbs_server* daemon and a job scheduler daemon are located on the head node. The batch job is submitted to the head node (in some cases, the job is first submitted to a login node and then transferred to the head node). A node list that records the configured compute nodes in the cluster is maintained on the head node. We briefly describe the procedure of job submission on TORQUE as follows:

1. The job is submitted to the head node by the command `qsub`. The job is normally written in a PBS (Portable Batch System) script. A job ID will be returned to the user as the standard output of `qsub`.
2. The job record, which incorporates a job ID and the job attributes, is generated and passed to *pbs_server*.
3. *pbs_server* transfers the job record to the job scheduler (e.g. Moab) daemon. The job scheduler daemon adds the job into a job queue and applies a scheduling algorithm to it (e.g. FIFO) which determines the job priority and its resource assignment.
4. When the scheduler finds the list of suitable nodes for the job, it returns the job information to *pbs_server*. The first node on this list becomes the *Mother superior* and the rest are called *sister MOMs* or *sister nodes*. *pbs_server* allocates the resources and passes the job control as well as execution information to the *pbs_mom* daemon installed on the mom superior node instructing to launch the job on the assigned compute nodes.
5. The *pbs_mom* daemons on the compute nodes manage the execution of jobs and monitors resource usage. *pbs_mom* will capture all the outputs and direct them to *stdout* and *stderr* into the output and

error files of the job and copy them to the designated location when the job completes successfully. The job status (completed or terminated) will be passed to *pbs_server* by *pbs_mom*. The job information will be updated.

Before and after a job execution, TORQUE executes *prologue* and *epilogue* scripts to prepare systems and perform node health check, append text to output and error log files, clean up system, etc.

In TORQUE, nodes are partitioned into different groups called *queues*. In each queue, the administrator sets limits for resources such as walltime and job size. This feature can be useful for job scheduling in a large HPC cluster where nodes are heterogeneous or certain nodes are reserved for special users.

Slurm

The structure of a Slurm [7] managed cluster is composed of one or two Slurm servers and many compute nodes. Figure 2 illustrates its structure. A Slurm server hosts the *slurmctld* daemon which is responsible for cluster resource and job management. Slurm servers and the corresponding *slurmctld* daemons can be deployed in active/passive mode in order to provide service of high reliability for computing clusters. Each compute node hosts one instance of the *slurmd* daemon, which is responsible for job staging and execution. There are additional daemons, e.g. *slurmdbd* that allows to collect and record accounting information for multiple Slurm-managed clusters and *slurmrestd* that can be used to interact with Slurm through a REST API. The Slurm resource list is held as a part of the *slurm.conf* file located on Slurm server nodes, which contains a list of nodes including features (e.g. CPU speed and model, amount of installed RAM) and configured partitions including partition names, list of associated nodes and priorities.

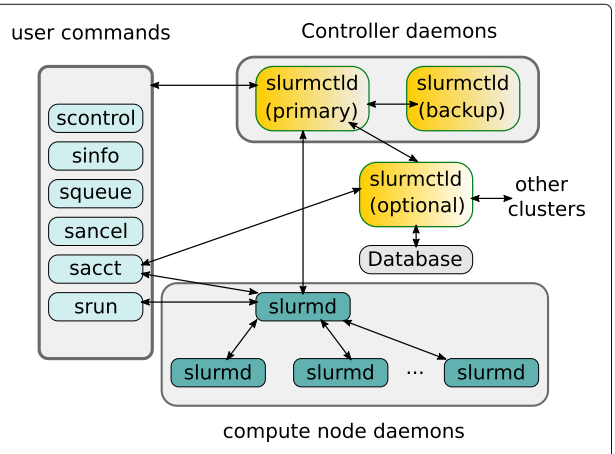


Fig. 2 Slurm structure [7]

Implementation of AI in HPC

The advancement of ML and Deep Learning (DL) have brought higher performance and more accurate solutions to applications in different domains, e.g. computer vision and natural language processing. ML and DL algorithms are often data intensive and computation demanding. Industry and academia seek the solutions in Cloud computing, as it allows large datasets to be easily ingested and managed to train algorithms, and enables ML and DL models to scale efficiently at low cost. Compared with Cloud, HPC systems show the advantages in computational power, storage and security [10]. Exploiting HPC infrastructures for ML and DL training is becoming a topic of increasing importance [11].

AI applications are usually developed with high level scripting languages or frameworks, e.g. TensorFlow [12] and Pytorch [13], which often require connections to external systems to download a list of open-source software packages during application execution. For example, an AI application written in Python cannot be compiled into an executable that includes all the dependencies necessary for execution as in C/C++, therefore, the developers need flexibility to customise their execution environments. Since HPC environments, especially on HPC production systems, are often based on closed source applications and their users have restricted account privileges and security restrictions [14], for instance the access to external systems is blocked. Deployment of AI applications on HPC infrastructure is challenging. Containerisation can be a potential candidate, which enables easy transition of AI workloads to HPC while fully taking advantage of HPC hardware and the optimised libraries of AI applications without compromising security on HPC systems.

Containerisation

A container is an OS level virtualisation technique that provides application execution environment separation. Figure 3 differentiates the architecture of VMs and containers. A traditional VM loads an entire guest OS into memory, which can occupy gigabytes of storage space on the host and requires a significant fraction of system resources to run. *Per contra*, a container can utilise the dependencies on its host OS. The host merely needs to start new processes to boot new containers [15], thus making start-up time of a container similar to that of a native application [16–22]. Apart from portability, containers can also guarantee reproducibility, i.e. once a workflow has been defined and stored in the container, its included working environment remains unaltered regardless of its running occurrences. Containers can run inside VMs as this is the case in most of the Cloud clusters [23].

There are multiple technologies that realise the concept of containers, e.g. Docker [24], Singularity [25], Shifter

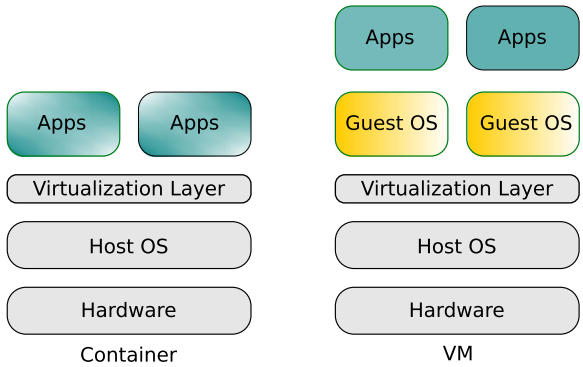


Fig. 3 Structure VMs and containers. On the VM side, the virtualisation layer often appears to be Hypervisor while on the container side it is the container runtime

[26], Charlie Cloud [27], Linux LXC [28] and Rkt Core OS [18]. Docker may be the most popular one on Cloud. Singularity developed for HPC systems shows the following merits:

- Run with user privileges and need no daemon process. Acquisition of root permission is only necessary when building images, which can be performed on user working computers;
- Seamless integration with HPC. Singularity natively supports GPU, Message Passing Interface (MPI) [29] and InfiniBand. In contrast with Docker, it demands no additional network configurations;
- Portable via a single image file. Au contraire, Docker is built up on top of layers of files.

Most Docker images can be converted to Singularity images. Singularity has thereby become the standard container runtime in practice for HPC systems.

HPC applications are often specifically optimised for the nodes, which is not the case for containerised applications. Furthermore, not all containerised applications can execute in parallel. Considering that performance is essential for HPC applications, it poses the key question on massive usage of containerised applications on HPC clusters [30].

Container orchestration for cloud clusters

The complex containerised applications in production can benefit immensely from container orchestrators that offer [31–33]:

- Resource limit control. This feature reserves the CPU and memory for a container, which restrains interference among containers and provides information for scheduling decisions;
- Scheduling. Determine the policies on how to optimise the placement of containers on specific nodes;
- Load balancing. Distribute the load among container instances;
- Health check. Verify if a faulty container needs to be destroyed or replaced;
- Fault tolerance. Create containers automatically if applications or nodes fail;
- Auto-scaling. Add or remove containers automatically.

Kubernetes [32] has a rapidly growing community and ecosystem with numerous platforms being developed upon it. Its architecture is composed of a master node and many worker nodes. Kubernetes runs its containers inside *pods* that are scheduled to the worker nodes. A *pod* can encapsulate one or multiple containers. Kubernetes provides its services by *deployment* that are created by submission of *yaml* files. In the *yaml* file, users specify the services and computation.

Kubernetes is based on a highly modular architecture which abstracts the underlying infrastructure and allows internal customisation, such as deployment of different software defined network or storage solutions. It supports various Big-Data frameworks (e.g. Hadoop MapReduce [34], Apache Spark [35] and Kafka [36]) and can be connected with Ansible [37] which is a widely-used software orchestration tool in Cloud clusters. Kubernetes includes a powerful set of tools to control the life cycle of applications, e.g. parameterised redeployment in case of failures, state management, etc. Furthermore, Kubernetes incorporates an advanced scheduling system which can even specify different schedulers for each job. Kubernetes supports software defined infrastructures[1] and resource disaggregation [38] by leveraging container-based deployment and particular drivers (e.g. Container Network Interface driver) based on standardised interfaces. These interfaces enable the definition of abstractions for fine-grain control of computation, states and communication in multi-tenant Cloud environment along with optimal usage of the underlying hardware resources.

Related work

This paper extends our work-in-progress study [39] that has briefly described the preliminary design of the Torque-Operator and platform architecture, which enables the convergence of HPC and Cloud systems. In [39], a simple testing case was also given to illustrate validation of the implementation. We herein extend the

[1] Software-defined infrastructure (SDI) is the definition of technical computing infrastructure entirely under the control of software with no operator or human intervention. It operates independent of any hardware-specific dependencies and is programmatically extensible.

description of the platform architecture and the structure of Torque-Operator. Furthermore, performance evaluation is demonstrated by three ML use cases.

A few studies [1, 40, 41] have been carried out on container orchestration for HPC clusters, since HPC systems are just starting to adopt the containerisation technology. Some works [39, 42, 43] have been performed on the general issues of bridging the gap between conventional HPC and service-oriented infrastructures. Nevertheless, literature has shown numerous works [31, 44–46] on container orchestration for Cloud clusters, though this is out of the scope for this paper.

Liu et al. [42] showed how to dynamically migrate computing resources between HPC and OpenStack clusters based on demand. At a higher level, IBM has demonstrated the ability to run Kubernetes *pods* on Spectrum LSF (an HPC workload manager).

Piras et al. [43] implemented a method that expanded Kubernetes clusters onto HPC clusters through Grid Engine (an HPC workload manager). Contrary to our work, submission is performed by the PBS job to launch Kubernetes jobs. Therefore, HPC nodes are added to Kubernetes clusters by installing Kubernetes core components (i.e. *kubeadm*, *Kubelet*) and Docker container runtime on HPC nodes. On HPC, especially HPC production systems, it is difficult to add new software packages as this can cause security concerns and alter working environment for current users.

Khan et al. [1] proposed to containerise HPC workloads and install Mesos [47] and Marathon on HPC clusters for resource management and container orchestration. Its orchestration system can obtain the appropriate resources satisfying the needs of requested services within defined Quality-of-Service (QoS) parameters, which is considered to be self-organised and self-managed meaning that users do not need to specifically request resource reservation. Nevertheless, this study has not shown insight into novel strategies of container orchestration for HPC systems.

Julian et al. [40] proposed their prototype for container orchestration in an HPC environment. A PBS-based HPC cluster that can automatically scale up and down as load demands by launching Docker containers using Moab scheduler. Three containers serve as front-end system, scheduler (it runs workload manager inside) and compute node (launches *pbs_mom* daemon). More compute node containers are scheduled when there are no sufficient number of physical nodes. Unused containers are destroyed via external Python scripts when jobs complete. Similarly, an early study [41] described two models that can orchestrate Docker containers using an HPC resource manager. The former model launches a container to mimic one compute node which holds all the processes assigned to, whilst the latter boots a container per process. However this work seems to be less appealing to HPC systems. MPI applications dominate HPC programs, which can be automatically scaled with Singularity runtime support.

Wrede et al. [48] performed their experiments on HPC clusters using Docker Swarm as the container orchestrator for automatic node scaling and using C++ algorithmic skeleton library Muesli [49] for load balance. Nevertheless, its proposed working environment is targeted for Cloud clusters. Usage of Docker cannot be easily extended to HPC infrastructures especially to HPC production systems due to the security concerns.

Architecture and tool description

Traditional HPC workload managers lack efficiencies in container scheduling and management, and often do not provide integrated support for environment provisioning (i.e. infrastructure, configuration and dependency). Table 1 compares the main differences between HPC workload managers and container orchestrators.

Cloud clusters demonstrate flexibility in environment customisation. The corresponding container orchestrators, such as Kubernetes, show their advantages in container management. In this section, we propose a platform architecture that bridges Cloud and HPC clusters, where users can customise the execution environment on the Cloud cluster to develop and deploy their service programs, meanwhile the compute-intensive or data-intensive jobs can be scheduled to HPC clusters where their performance could be significantly enhanced. The jobs are co-scheduled and co-managed by both Kubernetes and TORQUE. More specifically, on the first-level, the jobs are scheduled by Kubernetes and then by TORQUE on the second level.

Architecture and the testbed setting

The platform consists of an HPC cluster with TORQUE as its workload manager and a Cloud cluster with

Table 1 Comparison of HPC workload managers and container orchestrators

	HPC workload manager	Container orchestrator
Deployment	Batch queue (queueing time from seconds to days)	Often immediate
Workload type	Binary	Container, pod
Resource unit	Bare-metal nodes	Pods, VM nodes
Application execution length	Run to completion	Continuously running
Application specifics	Distributed memory jobs (e.g. MPI jobs)	Cloud micro services
DevOps environment provision	No	Yes

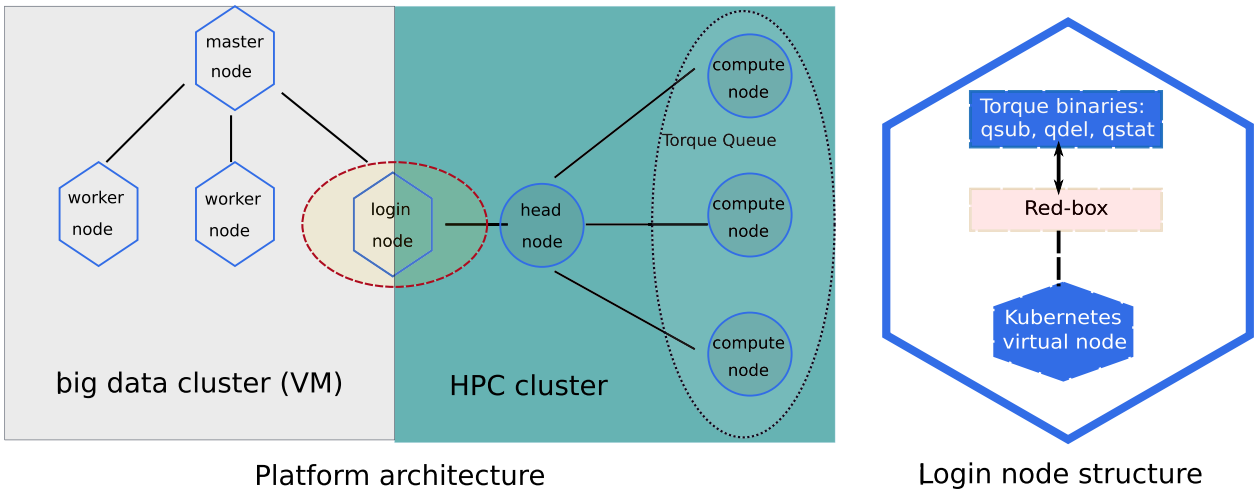


Fig. 4 Platform architecture. The login node belongs to both the Kubernetes and TORQUE clusters. The HPC cluster encompasses only one queue in the figure. The virtual node (inside the login node structure) on the right figure corresponds to the TORQUE queue on the left figure. Multiple virtual nodes can be generated when multiple TORQUE queues exist

Kubernetes as its container orchestrator. Its architecture is illustrated in Fig. 4 on the left. For simplicity, Fig. 4 shows a limited number of nodes and a single TORQUE queue. However, the architecture is not limited to this node number and can be extended to support more nodes and TORQUE queues. The only requirement here is the existence of one or more shared login nodes.

The HPC cluster is composed of a head node, compute nodes (execute workload) and login nodes. The login node highlighted in Fig. 4 plays the indispensable role of bridging the TORQUE and Kubernetes clusters. It is a VM node that serves as one of the worker nodes in the Kubernetes cluster (the Cloud cluster), meanwhile, it acts as a login node for TORQUE. The login node only submits the TORQUE job to the HPC cluster and is not included in the TORQUE compute node list. Job submission on the HPC cluster side will not be scheduled to this login node. The Kubernetes cluster incorporates a master node that schedules the jobs to the worker nodes. The work nodes deliver the services or perform computation. This architecture brings the following merits:

- Provide a unified interface for users to access the Cloud and the HPC clusters. Jobs are submitted in the form of *yaml*;
- Except Singularity, no additional software packages are needed to be installed in the HPC system. Hence it gives little impact on the working environment of existing HPC users;
- Users have flexibility to run containerised and non-containerised applications (on the HPC side);
- The performance of compute-intensive or data-intensive applications can be significantly improved via execution on the HPC cluster;
- Containers scheduled by Kubernetes to HPC clusters can take advantage of the container scheduling strategies of Kubernetes, where TORQUE lacks its efficiency.

Architecture of torque-operator

We developed a tool named *Torque-Operator* that bridges Kubernetes and TORQUE. Torque-Operator is an extension of WLM-Operator [50]. It connects with Kubernetes by creating a *deployment* called torque-operator on the Kubernetes cluster as shown in Fig. 5. The tool is written in the Golang programming language. A virtual node (named *torque-sycri-k8s-2-batch* in Fig. 6) is created, which corresponds to one TORQUE queue. Figure 6 shows the virtual node information. *Virtual node* is a concept in Kubernetes. It is not a real worker node, however, it contains the information of its corresponding HPC queue (e.g. the number of nodes per queue) and can schedule the jobs to real worker nodes. The virtual node connects Kubernetes to other APIs and allows developers to deploy *pods* and containers with their own APIs. The number of virtual nodes can be multiple when more than one queue exists on the TORQUE cluster. The services provided by the *deployment* are carried out by four Singularity containers which:

```
$kubectl get deployments
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
torque-operator   0/1      1            0          11d
```

Fig. 5 The *deployment* of Torque-Operator. The *deployment* has been running continuously to provide services

```
$kubectl get nodes --show-labels
NAME                        STATUS  ROLES   AGE   VERSION          LABELS
torque-sycri-k8s-2-batch    Ready   agent   11d   v1.13.1-vk-N/A   ...,wlm.sylabs.io/nodes=2,...
```

Fig. 6 The virtual node generated by Torque-Operator. Only one node label is displayed and the rest is omitted in the figure for simplicity

- generate the virtual node;
- fetch the queue information of the TORQUE cluster and add the information to the virtual node as its *node label*. A node label indicates the restrictions of a node. The jobs that meet such restrictions can be scheduled to the node. For example, the label of the virtual node (*wlm.sylabs.io/nodes* = 2 in Fig. 6) denotes that the TORQUE cluster contains 2 compute nodes, thus a Kubernetes job which requests more than 2 compute nodes will be in a pending status;
- launch TORQUE jobs to the Kubernetes cluster by *pods*;
- transfer the TORQUE jobs to the TORQUE cluster and return the results obtained on the TORQUE cluster to the directory which is specified in the *yaml* file submitted by users as indicated in Fig. 7 Line 20.

Besides the four containers, Torque-Operator also includes a Linux service program named *red-box* running on the login node as shown already in Fig. 4. Redbox builds a Unix socket which allows data exchange among the Kubernetes and TORQUE processes. Torque-Operator introduces a new Kubernetes *object kind i.e. Torquejob*. As users can use `kubectl get pods` to display the *pod* information, they can perform `kubectl get torquejob` to show the status of jobs submitted to TORQUE. Users can also view the status of TORQUE jobs using the PBS commands `qstat` on the login node (marked in red dashed line in Fig. 4).

```
1  apiVersion: wlm.sylabs.io/v1alpha1
2  kind: TorqueJob
3  metadata
4     name: cow
5  spec:
6     batch: |
7     #!/bin/sh
8     #PBS -l walltime=00:30:00
9     #PBS -l nodes=1
10    #PBS -e $HOME/low.err
11    #PBS -o $HOME/low.out
12    export PATH=$PATH:/usr/local/bin
13    singularity pull -U library://sylabsed/examples/low
14    singularity run lolcow_latest.sif
15 results:
16    from: $HOME/low.out
17    mount:
18      name: data
19      hostPath:
20        path: $HOME/
21        type: DirectoryOrCreate
```

Fig. 7 An example of the Kubernetes *yaml* script. The script encloses a PBS script. Submission is performed as a normal *yaml* job submission, i.e. `$kubectl apply -f $HOME/cow_job.yaml`

The PBS job script is encapsulated into a Kubernetes *yaml* job script. The *yaml* script is submitted from the login node. An example of the *yaml* script named `cow_job.yaml` is illustrated in Fig. 7. The PBS job is included from Line 7 to Line 14. More precisely, the job requests 30 minutes walltime and one compute node. It executes a Singularity image called `lolcow.sif` located in the `home` directory, which is pulled from Syslab registry. Its results and error messages will be written in `low.out` and `low.err`, respectively. Line 12 sets the necessary `PATH` environment variable to find the Singularity executable located in `/usr/local/bin` on the HPC system.

The TORQUE job script part is abstracted by Toque-Operator. A Kubernetes *pod* is generated to transfer the PBS job specification to a scheduling queue of the HPC cluster (e.g. waiting queue or test queue as in TORQUE job scheduler, while the concept of *queue* in Fig. 4 is the partition of nodes). Torque-Operator invokes the TORQUE binary `qsub` that submits the job to the TORQUE cluster. When the job completes, Torque-Operator creates a Kubernetes *pod* which redirects the results to the directory that the user specifies.

The *yaml* job submitted from the Kubernetes login node is scheduled by the Kubernetes master node to the virtual node. The virtual node transfers the abstracted jobs to the TORQUE queue through the login node. Figure 8 illustrates the main procedures of container job submission and execution from the Kubernetes cluster to the TORQUE cluster.

User permission management

With WLM-Operator [50], all submitted jobs will be executed on the HPC cluster on behalf of a single user. This gives performance limitation and causes security concern especially in an HPC production system, as multiple users may submit jobs simultaneously and their data confidentiality cannot be guaranteed. Furthermore, we need multi-user support in order to enable individual monitoring, accounting, fair-share scheduling and other features per single user or per group of users which are supported by default on TORQUE. This is achieved in Torque-Operator by providing a dynamic adaptation of the user context through automatically reconfiguring the Kubernetes virtual node along with the corresponding red-box socket (see Fig. 4) to use the right user privileges. The current reconfiguration mechanism [51] brings multi-user support that addresses the confidentiality issues, though

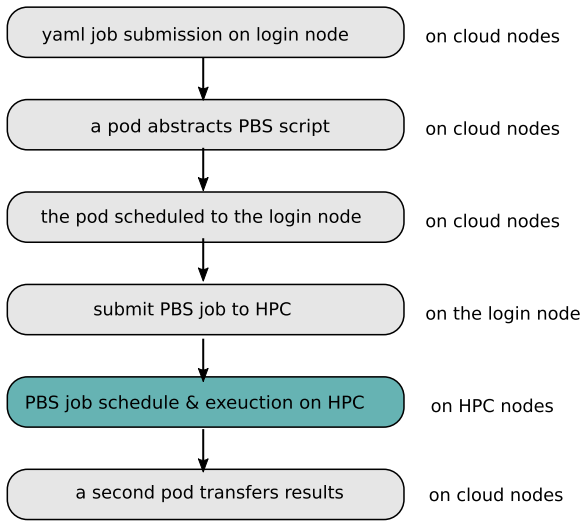


Fig. 8 The workflow of job execution procedure

it still lacks the capability of concurrent submissions from multiple users. We propose to add this feature to a future release of Torque-Operator.

Use cases

The proposed architecture and tool herein are implemented in the testbed of the EU research project CYBELE[2]. We present two use cases from the CYBELE project for functionality validation and performance evaluation. The two use cases herein are named **Pilot Wheat Ear** and **Pilot Soybean Farming**, respectively. In addition, we give the performance evaluation on an open-source MPI benchmark: **BPMF** [3] ("Pilot and benchmark description" section). All the three applications are containerised in Singularity. Table 2 lists the key software packages and their dependencies encapsulated in Singlarity images for the three applications.

It is straightforward to build a Singularity image which is as easy as installing software packages in a Linux system. We illustrate a snapshot of the Singularity build script for **Pilot Soybean Farming** in Fig. 9 as an example. The other two scripts carry the same principle. In Fig. 9, the image is based on Ubuntu image version 18.04 as indicated in Line 2. Line 4 (`%environment`) starts the section on definition of the environment variables that will be set at runtime (from Line 5 to Line 8). Commands in the `%post` (Line 10) section are executed after the base OS (i.e. Ubuntu 18.04) has been installed at build time,

[2]CYBELE: Fostering Precision Agriculture and Livestock Farming through Secure Access to Large-Scale HPC-Enabled Virtual Industrial Experimentation Environment Empowering Scalable Big Data Analytics. https://www.cybele-project.eu/

[3]https://github.com/ExaScience/bpmf/

Table 2 The key software packages and their dependencies packed in Singularity images for each application

Use cases	Software packages and dependencies
Pilot Wheat Ear	Pytorch, Python,libjpeg,libpng-dev,libnccl2, libibverbs,libnuma,librdmacm,libmlx4,libmlx5
Pilot Soybean Farming	Python, Numpy, Pandas, gdal, scipy, scikit-learn, openpyxl, xlrd
BPMF	Open MPI

Software versions are omitted in the table

more specifically, from Line 11, all the necessary software packages and dependencies are installed.

Pilot and benchmark description

Pilot Wheat Ear is targeted for provisioning of autonomous robotic systems within arable frameworks. More specifically, the application provides a framework

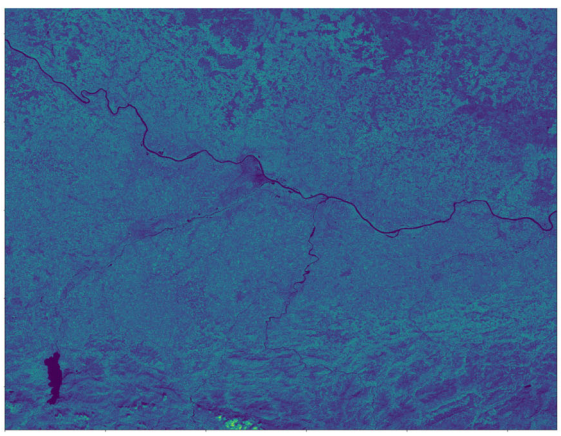
```
1  Bootstrap: docker
2  From: ubuntu:18.04
3
4  %environment
5  export LC_ALL=C
6  export PYTHON_VERSION=3.7
7  export CPLUS_INCLUDE_PATH=/usr/include/gdal
8  export C_INCLUDE_PATH=/usr/include/gdal
9
10 %post
11 apt-get -y update&& apt-get install -y \
12 build-essential \
13 libssl-dev \
14 openssl \
15 libreadline-dev \
16 software-properties-common \
17 cmake
18 add-apt-repository ppa:ubuntugis/ppa \
19 && apt-get -y update
20 add-apt-repository ppa:deadsnakes/ppa
21 apt-get -y update
22 apt-get install -y \
23 python3.7 \
24 python3.7-dev \
25 python3.7-venv \
26 libatlas-base-dev
27 apt-cache policy gdal-bin
28 #prepare gdal
29 apt-get install -y \
30 gdal-bin=2.2.2+dfsg-1~xenial1 \
31 libgdal-dev
32 export CPLUS_INCLUDE_PATH=/usr/include/gdal
33 export C_INCLUDE_PATH=/usr/include/gdal
34 #install python pip
35 apt-get install -y python3-pip
36 pip3 install --upgrade setuptools
37 #install numpy
38 yes w | python3.7 -m pip install numpy
39  ...
```

Fig. 9 A snapshot of the Singularity build script for **Pilot Soybean Farming**

for automatic identification and count of wheat ears in fields from the data collected by sensors on ground so that it enables crop yield prediction at early stages and can suggest decisions for sales planning. The application trains a DL algorithm written in Python on Fastai/Pytorch [52][13] framework based on a series of RGB images (138 images, 95 MB in total). An example of the images is illustrated in Fig. 10a.

Pilot Soybean Farming focuses on the application of ML in soybean farming. Its goal is to develop a prediction algorithm that is able to infer hidden dependencies between the input parameters and the yield. The ML algorithm, more specifically the Random Forest algorithm [53], is implemented in Python's Scikit learn library [54]. The training dataset consists of two Sentinel-2 [55] images (the total size of images: 4.9 GB) with 13 spectral bands, at a resolution of 10m and yield monitor data (total size: 569 MB) from Austrian soybean farms. The pipeline includes modules for image stacking, cropping to the field boundaries, yield map interpolation from point measurements and a Random Forest module for yield prediction. An example of the Sentinel-2 image (only a single band of the image is displayed) is shown in Fig. 10b.

(a) **A Wheat Ear image.**



(b) **A Sentinel-2 image. A single band of the image is shown.**

Fig. 10 Examples of Pilot images. The Sentinel-2 image deliver information of three key stages of soya growth

The aforementioned pilots are in developing stage and are not yet publicly accessible. The former can only scale on the cores of one single node while the latter only executes on a single core of one node. To demonstrate the performance improvement that the HPC cluster can bring and illustrate that the approaches introduced herein can be applied in more general cases, we present performance evaluation on an MPI benchmark **BPMF** [56, 57]. The **BPMF** benchmark adopts the ML algorithm, i.e. the Bayesian Probabilistic Matrix Factorization (BPMF) method, to predict compound-on-protein activity using Markov Chain Monte Carlo. It uses a Gibbs sampler and works with MPI and Eigen library handling the linear algebra and related algorithms. Each sampling iteration consists of two distinct regions on compounds and protein activities. The number of sampling iterations is set to be 100 in this experiment. MPI [29, 58] applications as the conventional parallel programs for HPC are well designed to scale on multiple or many cores. Table 3 summarises the descriptions of the two pilots and the MPI benchmark.

Two approaches are often used to execute MPI applications using Singularity, i.e. hybrid model and bind model. The former compiles MPI binaries, libraries and the MPI application into a Singularity image. The latter binds the container on a host location where the container utilises the MPI libraries and binaries on the host HPC systems. The latter model has smaller image size since it does not include compiled MPI libraries and binaries in the container image. However, the latter has some drawbacks:

- The MPI version that is used to compile the application in the container must be compatible with the version of MPI available on the host;
- Users must know where the host MPI is installed;
- User must ensure that binding the directory where the host MPI is installed is possible.

Table 3 Descriptions of the pilots and **BPMF**

Names	Description	ML applications
Pilot Wheat Ear	Provision of autonomous robotic systems within arable frameworks	Yes
Pliot Soybean Farming	Yield prediction for soybean farming	Yes
BPMF	MPI bechmark: predict compound-on-protein activity	Yes

Table 4 Technical Specifications of the testbed

Cluster name	HPC cluster (bare-metal)	VM cluster
Total number of nodes	3 (2 compute nodes)	4 (3 worker nodes)
Number of cores per node	20 (10 cores per CPU, 2 CPU per node)	2
RAM per node (NUMA)	128 GB	7.79 GB
CPU frequency	Intel(R)Xeon(R) CPU E5-2630 v4, 2.20GHz	Intel i7 9xx (Nehalem Core i7, IBRS) 2.79GHz
Operating System	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS
VM type	-	QEMU 2.11.1, KVM 2.11.1

We chose the hybrid model as mounting storage volumes on the host as in the bind model can require privileged operations. In the hybrid model, the MPI launcher of the host system invokes Singularity container and Singularity launches the MPI application within the container. We use Open MPI [59] with version v4.0.4 as the MPI library.

Performance evaluation

Performance evaluation is carried out on the testbed with the technical specifications indicated in Table 4. Singularity is the container runtime of our choice, as it provides a secure means to capture and distribute software and their environments. Kubernetes supports Docker by default, though it can be adjusted to support services for Singularity by adding Singularity-CRI [4] [60]. Table 5 concludes the list of core applications to be installed on the testbed. In general, Kubernetes cluster can support both the Singularity runtime and the Docker runtime. Ideally the worker nodes which run Docker containers should not be linked to HPC clusters. Romana is deployed as the network model for the container network interface (CNI) of Kubernetes. Without this additional network model, the *pods* located on different hosts could not establish communication with each other. Romana [61] is an open source network and security automation solution that enables deployment of Kubernetes without an overlay network. It supports Kubernetes Network Policy to provide isolation across network namespaces.

Figure 11 compares the performance difference for **Pilot Wheat Ear**, **Pilot Soybean Farming** and **BPMF** runing on the Cloud cluster (VM) and HPC cluster. On the VM cluster, the applications only execute with 2 processes as it is the maximal number of the VM cores on one node. The performance ranges from execution with 2 processes

[4]Singularity-CRI: it is Singularity-specific implementation of Kubernetes CRI (CRI: Container Runtime Interface).

Table 5 The list of core applications for the testbed

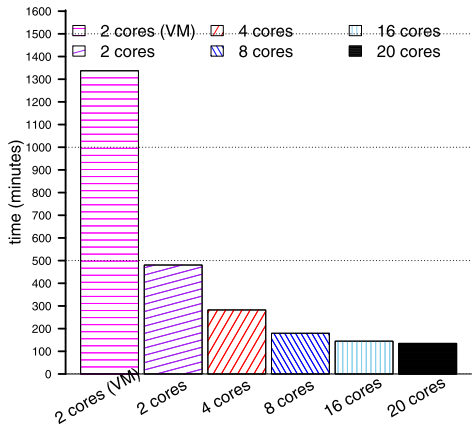
Cluster types	Cloud cluster	HPC cluster
Orchestrator	Kubernetes	Torque
Container runtime & interface	Singularity, Singulairy-CRI	Singularity
Plugin	Torque-Operator, Romana	-
Compiler	Golang compiler	Golang compiler

to 20 processes as the maximal number of the cores on one HPC node. We only present the results of **Pilot soybean Farming** running on one core as it is a sequential program. The performance details are given in Table 6. The results are stable, which are the average of three executions. The variances of execution time are shown in the table in brackets. Figure 11 indicates that the performance of the three applications has been significantly improved when running on the HPC cluster comparing with that on the VM node (by 64.06%, 48.45% and 38.20% running on two cores). Execution time of **Pilot Wheat Ear** and **BPMF** on the HPC cluster are further shortened with incrementing core numbers. The time spent in job scheduling from Cloud to HPC is negligible compared to the total execution time. The potential immense gain for the out-of-the-box performance is one major reason for levaraging HPC to AI applications, while usage of containers make deployment of AI applications portable on HPC clusters.

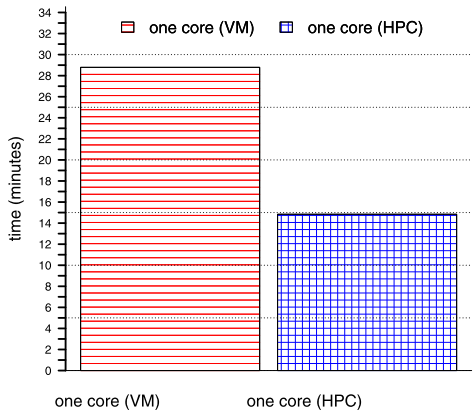
Discussion

It is difficult to migrate applications from Cloud to HPC clusters. Although containerising applications can meet the portability requirements, their performance is not always satisfying. HPC clusters demonstrate their advantages by running applications concurrently on many cores and nodes. Obviously the performance of applications on bare-metal HPC nodes surpasses that on VM nodes, even though VM nodes are configured with more powerful CPU in the testbed. However, the programs which do not scale benefit less from running on HPC systems, as this is the case for **Pilot Soybean Farming** that is a sequential program.

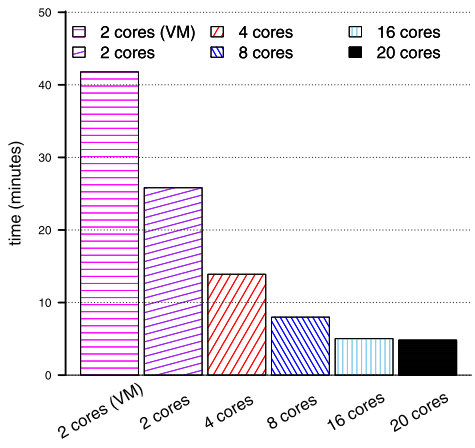
HPC applications tend to scale on many cores. This article intends to propose a methodology which can provide a common interface between HPC and Cloud clusters enabling container orchestration on HPC systems. This structure is particularly important for AI applications that tend to be compute-intensive or data-intensive, thereby, can benefit immensely from HPC clusters. Scalability of AI applications on HPC nodes should be taken into account. This can be addressed by containerising the DL Framework Horovod [62] (Horovod adopts



(a) **Pilot Wheat Ear**



(b) **Pilot Soybean Farming**



(c) **BPMF**

Fig. 11 Time comparison for the two pilots and **BPMF** running on the Cloud node and HPC node

Table 6 Execution time on the Cloud and HPC cluster for 3 use cases on a single node (unit: minutes)

Cores (on 1 node)	Pilot Wheat Ear	Pilot Soybean Farming	BPMF
Cloud (VM) 1 core	-	28.79 (1.87)	-
Cloud (VM) 2 cores	1337.13 (0.00)	-	41.78 (0.03)
HPC (bare-metal) 1 core	-	14.84 (0.02)	-
HPC (bare-metal) 2 cores	480.50 (4.97)	-	25.82 (0.01)
HPC (bare-metal) 4 cores	282.39 (0.28)	-	13.90 (0.00)
HPC (bare-metal) 8 cores	179.54 (1.02)	-	7.99 (0.00)
HPC (bare-metal) 16 cores	144.76 (0.12)	-	5.03 (0.00)
HPC (bare-metal) 20 cores	134.99 (0.55)	-	4.84 (0.01)

The numbers in the brackets are the mathematical variances. Execution time of the two Pilots can vary from minutes to days depending on the amount of input data sets that are used for processing

MPI concepts, which enables massive HPC node scaling) into Singularity images. Alternatively, Spark Engine (Spark provides data parallelism) can be encapsulated into Singularity. In this case, Spark jobs will be scheduled by HPC workload managers such as TORQUE or Slurm in preference to big-data schedulers, e.g. Apache Hadoop YARN [63] or Mesos. However, it is out of scope of this paper to discuss parallelisation for containerised AI applications.

It is worth noting that our proposed architecture (Fig. 4) can be flexible, i.e. job submission is not restricted to the login node. Users can submit their jobs from the rest of the Kubernetes nodes which are connected with this login node in the same Kubernetes cluster. The master node will schedule the submission to the login node and the TORQUE job will be transferred to the HPC cluster. On our architecture, the login node which connects the Cloud and HPC systems are located in the two network domains: the domains of HPC cluster and Cloud cluster. On an HPC production system, a more portable and secure approach is to connect the login node remotely to the HPC cluster via `ssh`, although this could cause performance degradation when there is a large amount of data transmission.

Conclusion and future work

This article described an architecture and the structure of Torque-Operator. The architecture creates a synergy between HPC and Cloud. It provides users with flexibility to run containerised and non-containerised

jobs with the same submission interface. Compute-intensive or data-intensive jobs can gain higher performance by scheduling from the Cloud cluster to the HPC cluster with the scheduling performed by both Kubernetes and TORQUE. Two use cases of the EU research project CYBELE are presented to demonstrate the merits of our design. Nevertheless, the proposed architecture is not limited to the usages of our project pilots. We generalised its application by showcasing an open-source MPI benchmark containerised in Singularity.

In future work, it could be interesting to compare efficiency of container orchestration of our hybrid architecture with container scheduling performed by TORQUE only. This will hugely depend on the development of the AI use cases of the CYBELE project. Currently, container orchestration policies rely on Kubernetes. The next steps can be carried out on improvement of the container scheduling of TORQUE. In addition, the login node can accept the regular Kubernetes *yaml* file instead of embedding a TORQUE PBS script within it, and the orchestrator is able to analyse the job size so that it can determine whether to submit jobs to HPC clusters or schedule them to the Cloud clusters. In the latter case, the embedded PBS *yaml* will be automatically generated for HPC job submission.

Abbreviations

HPC: High performance computing; VM: Virtual machine; WLM: Workload manager; TORQUE: Terascale open-source resource and QUEue manager; OS: Operating system; PBS: Portable batch system; FIFO: First in first out; BPMF: Bayesian probabilistic matrix factorization

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement NO.825355.

Yield monitor data in Pilot Soybean Farming have been generously granted by Donau Soya association from farms across the Lower Austria region, through CYBELE project and are subject to NDA.

The authors would like to express the gratitude to Dr. Joseph Schuchart for proof-reading the contents.

Authors' contributions

All authors read and approved the final manuscript.

Funding

Open Access funding enabled and organized by Projekt DEAL.

Availability of data and materials

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Stuttgart, Germany. [2]Ryax Technologies, Lyon, France. [3]Institute of Bioorganic Chemistry of the Polish Academy of Sciences, Poznan Supercomputing and Networking Center (PSNC), Poznan, Poland. [4]BioSense Institute, University of Novi Sad, Novi Sad, Serbia.

Received: 16 October 2020 Accepted: 27 January 2021
Published online: 22 February 2021

References

1. Khan M, Becker T, Kuppuudaiyar P, Elster AC (2018) Container-Based Virtualization for Heterogeneous HPC Clouds: Insights from the EU H2020 CloudLightning Project. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE, Piscataway. pp 392–397
2. Rodriguez MA, Buyya R (2019) Container-based cluster orchestration systems: A taxonomy and future directions. Softw Pract Experience 49(5):698–719. https://doi.org/10.1002/spe.2660
3. Abdollahi Vayghan L, Saied MA, Toeroe M, Khendek F (2018) Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, Piscataway. pp 970–973
4. Casalicchio E (2017) Autonomic Orchestration of Containers: Problem Definition and Research Challenges. In: Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools. VALUETOOLS16. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL. pp 287–290. https://doi.org/10.4108/eai.25-10-2016.2266649
5. Hovestadt M, Kao O, Keller A, Streit A (2003) Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In: Feitelson D, Rudolph L, Schwiegelshohn U (eds). Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg, Berlin. pp 1–20
6. Klusáček D, Chlumský V, Rudová H (2015) Planning and Optimization in TORQUE Resource Manager. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. Association for Computing Machinery, New York. https://doi.org/10.1145/2749246.2749266
7. Jette MA, Yoo AB, Grondona M (2002) SLURM: Simple Linux Utility for Resource Management. In: In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003. Springer-Verlag, Berlin. pp 44–60
8. Staples G (2006) TORQUE Resource Manager. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. Association for Computing Machinery, New York. https://doi.org/10.1145/1188455.1188464
9. Moab HPC Suite. https://support.adaptivecomputing.com/wp-content/uploads/2019/06/Moab-HPC-Suite_datasheet_20190611.pdf. Accessed 08 July 2020
10. Mateescu G, Gentzsch W, Ribbens CJ (2011) Hybrid Computing-Where HPC Meets Grid and Cloud Computing. Future Gener Comput Syst 27(5):440–453. https://doi.org/10.1016/j.future.2010.11.003
11. Mayer R, Jacobsen HA (2020) Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools. ACM Comput Surv 53(1). https://doi.org/10.1145/3363554
12. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. (2016) TensorFlow: A System for Large-scale Machine Learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI'16. USENIX Association, Berkeley. pp 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899
13. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. (2019) PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R (eds). Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver. pp 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html
14. Brayford D, Vallecorsa S, Atanasov A, Baruffa F, Riviera W (2019) Deploying AI Frameworks on Secure HPC Systems with Containers. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, Piscataway. pp 1–6
15. Hale JS, Li L, Richardson CN, Wells GN (2017) Containers for Portable, Productive, and Performant Scientific Computing. Comput Sci Eng 19(6):40–50
16. Felter W, Ferreira A, Rajamony R, Rubio J (2015) An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE

International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, Piscataway. pp 171–172
17. Bernstein D (2014) Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Comput 1(3):81–84
18. Martin JP, Kandasamy A, Chandrasekaran K (2018) Exploring the Support for High Performance Applications in the Container Runtime Environment. Hum-Centric Comput Inf Sci 8(1). https://doi.org/10.1186/s13673-017-0124-3
19. Plauth M, Feinbube L, Polze A (2017) A Performance Survey of Lightweight Virtualization Techniques. In: De Paoli F, Schulte S, Broch Johnsen E (eds). Service-Oriented and Cloud Computing. Springer International Publishing, Cham. pp 34–48
20. Zhang J, Lu X, Panda DK (2017) Is Singularity-Based Container Technology Ready for Running MPI Applications on HPC Clouds? In: Proceedings of The10th International Conference on Utility and Cloud Computing. UCC 17. Association for Computing Machinery, New York. https://doi.org/10.1145/3147213.3147231
21. Hu G, Zhang Y, Chen W (2019) Exploring the Performance of Singularity for High Performance Computing Scenarios. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, Piscataway. pp 2587–2593
22. Younge AJ, Pedretti K, Grant RE, Brightwell R (2017) 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). IEEE, Piscataway. pp 74–81
23. VMWare (2018) Containers on Virtual Machines or Bare Metals?. VMware, Inc., Palo Alto. https://assets.contentstack.io/v3/assets/blt58b49a8a0e43b5ff/blta366cfae83d85681/5c742ba62617ffd7604a143c/vmwwp-containers-on-vms.pdf
24. Merkel D (2014) Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux J 2014(239):76–90
25. Kurtzer GM, Sochat VV, Bauer M (2017) Singularity: Scientific containers for mobility of compute. In: PloS one. PLOS, San Francisco
26. Gerhardt L, Bhimji W, Canon S, Fasel M, Jacobsen D, Mustafa M, et al. (2017) Shifter: Containers for HPC. J Phys Conf Ser 898:082021. https://doi.org/10.1088%2F1742-6596%2F898%2F8%2F082021
27. Priedhorsky R, Randles T (2017) Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC 17. Association for Computing Machinery, New York. https://doi.org/10.1145/3126908.3126925
28. S SK (2017) Practical LXC and LXD: Linux Containers for Virtualization and Orchestration. 1st ed. Apress, USA
29. Gropp W, Lusk E, Skjellum A (1994) Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press, Cambridge
30. Xavier MG, Neves MV, Rossi FD, Ferreto TC, Lange T, De Rose CAF (2013) Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, Piscataway. pp 233–240
31. Casalicchio E (2019) Container Orchestration: A Survey. In: Puliafito A (ed). Systems Modeling: Methodologies and Tools. Springer International Publishing, Cham. pp 221–235. https://doi.org/10.1007/978-3-319-92378-9_14
32. Hightower K, Burns B, Beda J (2017) Kubernetes: Up and Running Dive into the Future of Infrastructure. 1st ed. OReilly Media, Inc., Sebastopol
33. Casalicchio E, Iannucci S (2019) The state-of-the-art in container technologies: Application, orchestration and security. Concurrency Comput Pract Experience 32(17):e5668. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5668
34. Pandey S, Tokekar V (2014) Prominence of MapReduce in Big Data Processing. In: 2014 Fourth International Conference on Communication Systems and Network Technologies. IEEE, Piscataway. pp 555–560
35. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, et al. (2016) Apache Spark: A Unified Engine for Big Data Processing. Commun ACM 59(11):56–65. http://doi.acm.org/10.1145/2934664
36. Narkhede N, Shapira G, Palino T (2017) Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale. 1st ed. O'Reilly Media, Inc., Sebastopol
37. Sammons G (2016) Exploring Ansible 2: Fast and Easy Guide. CreateSpace Independent Publishing Platform, North Charleston
38. Gao PX, Narayan A, Karandikar S, Carreira J, Han S, Agarwal R, et al. (2016) Network Requirements for Resource Disaggregation. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI|16. USENIX Association, USA. pp 249–264
39. Zhou N, Georgiou Y, Zhong L, Zhou H, Pospieszny M (2020) Container Orchestration on HPC Systems. In: 2020 IEEE International Conference on Cloud Computing (CLOUD). IEEE, Piscataway
40. Julian S, Shuey M, Cook S (2016) Containers in Research: Initial Experiences with Lightweight Infrastructure. In: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale. XSEDE16. Association for Computing Machinery, New York. https://doi.org/10.1145/2949550.2949562
41. Higgins J, Holmes V, Venters C (2015) Orchestrating Docker Containers in the HPC Environment. In: Kunkel JM, Ludwig T (eds). High Performance Computing. Springer International Publishing, Cham. pp 506–513
42. Liu F, Keahey K, Riteau P, Weissman J (2018) Dynamically Negotiating Capacity between On-Demand and Batch Clusters. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. SC 18. IEEE Press, Piscataway
43. Piras ME, Pireddu L, Moro M, Zanetti G (2019) Container Orchestration on HPC Clusters. In: Weiland M, Juckeland G, Alam S, Jagode H (eds). High Performance Computing. Springer International Publishing, Cham. pp 25–35
44. Fernandez GP, Brito A (2019) Secure Container Orchestration in the Cloud: Policies and Implementation. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. SAC 19. Association for Computing Machinery, New York. pp 138–145. https://doi.org/10.1145/3297280.3297296
45. Maenhaut PJ, Volckaert B, Ongenae V, De Turck F (2019) Resource Management in a Containerized Cloud: Status and Challenges. J Netw Syst Manag 28:197–246
46. Buyya R, Srirama SN (2019) A Lightweight Container Middleware for Edge Cloud Architectures. Wiley Telecom. https://ieeexplore.ieee.org/document/8654087
47. Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, et al. (2011) Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. NSDI 11. USENIX Association, USA. pp 295–308
48. Wrede F, von Hof V (2017) Enabling Efficient Use of Algorithmic Skeletons in Cloud Environments: Container-Based Virtualization for Hybrid CPU-GPU Execution of Data-Parallel Skeletons. In: Proceedings of the Symposium on Applied Computing. SAC 17. Association for Computing Machinery, New York. pp 1593–1596. https://doi.org/10.1145/3019612.3019894
49. Ciechanowicz P, Poldner M, Kuchen H (2009) The Münster Skeleton Library Muesli: A comprehensive overview. University of Münster, European Research Center for Information Systems (ERCIS). Available from: https://www.ercis.org/sites/www.ercis.org/files/pages/research/ercis-working-papers/ercis_wp_07.pdf
50. Pisaruk V, Yakovtseva S WLM-operator. Gitlab. https://github.com/sylabs/wlm-operator. Accessed 13 Feb 2020
51. Georgiou Y, Zhou N, Zhong L, Hoppe D, Pospieszny M, Papadopoulou N, et al. (2020) Converging HPC, Big Data and Cloud technologies for precision agriculture data analytics on supercomputers. In: 15th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'20). Springer International Publishing, Cham
52. Howard J, Gugger S (2020) Fastai: A Layered API for Deep Learning. Information 11(2):108. https://doi.org/10.3390/info11020108
53. Breiman L (2001) Random forests. Mach Learn 45(1):5–32
54. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. (2011) Scikit-Learn: Machine Learning in Python. J Mach Learn Res 12:2825–2830
55. Drusch M, Bello UD, Carlier S, Colin O, Fernandez V, Gascon F, et al. (2012) Sentinel-2: ESA's Optical High-Resolution Mission for GMES Operational Services. Remote Sens Environ 120:25–36. The Sentinel Missions - New Opportunities for Science. http://www.sciencedirect.com/science/article/pii/S0034425712000636

56. Salakhutdinov R, Mnih A (2008) Bayesian probabilistic matrix factorization using Markov chain Monte Carlo. In: Proceedings of the International Conference on Machine Learning. vol. 25. Association for Computing Machinery, New York
57. Aa TV, Chakroun I, Haber T (2016) Distributed Bayesian probabilistic matrix factorization. In: CLUSTER. IEEE Computer Society, Piscataway. pp 346–349
58. MPI: A Message-Passing Interface Standard. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Accessed 26 Jan 2021
59. Graham RL, Woodall TS, Squyres JM (2005) Open MPI: A Flexible High Performance MPI. In: Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics. PPAM 05. Springer-Verlag, Berlin. pp 228–239. https://doi.org/10.1007/11752578_29
60. Sylabs Singularity-CRI. https://sylabs.io/guides/cri/1.0/user-guide/k8s.html. Accessed 03 Mar 2020
61. Romana. https://romana.io/. Accessed 21 May 2020
62. Sergeev A, Balso MD (2018) Horovod: fast and easy distributed deep learning in TensorFlow. CoRR. abs/1802.05799. Available from: https://arxiv.org/abs/1802.05799
63. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, et al. (2013) Apache Hadoop YARN: Yet Another Resource Negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing. SOCC '13. Association for Computing Machinery, New York. https://doi.org/10.1145/2523616.2523633

Publisher's Note