Interactive, Cloud-Native Workflows on HPC Using KNoC

Evangelos Maliaroudakis[1,2(✉)], Antony Chazapis[1], Alexandros Kanterakis[1], Manolis Marazakis[1], and Angelos Bilas[1,2]

[1] Institute of Computer Science, FORTH, Heraklion, Greece
{malvag,chazapis,kantale,maraz,bilas}@ics.forth.gr
[2] Computer Science Department, University of Crete, Heraklion, Greece

Abstract. Cloud and HPC platforms differentiate by many aspects, but both can run applications in identical contexts using containers. In this paper we present KNoC, an open-source virtual node (kubelet) for Kubernetes that transparently manages the container lifecycle on a remote HPC cluster using Slurm and Singularity. Our goal is on one hand to allow HPC users to leverage existing cloud-native tools, such as the popular Argo Workflows language to express complex data-processing pipelines, while on the other hand enabling Cloud setups to exploit computing resources available in HPC centers. KNoC bridges Cloud and HPC, transforming Argo to a cross-environment, portable solution, which allows the combination of Cloud-based tools and HPC steps into the same workflow, controlled and monitored through an interactive frontend. Deploying KNoC requires only a secure shell connection to the cluster's login node. We describe the design and implementation of KNoC, and evaluate the integration using several proof-of-concept workflows.

Keywords: Cloud-HPC convergence · Reproducible workflows · Kubernetes extensions · Virtual kubelet

1 Introduction

As we are gradually transitioning into the *exascale* era, there is no shortage of infrastructure to process large datasets: the Cloud provides an abundance of storage and computing resources, while High-Performance Computing (HPC) facilities around the globe are constantly powering on bigger and more powerful machines, each combining thousands of general-purpose and domain-specific processing units [16]. The increasing complexity of applications shifts the developers' focus to higher-level, more expressive languages, while the heterogeneous computing landscape places the emphasis on portability and integration issues. Large processing pipelines should ideally be synthesized as portable workflows that can move between setups, enabling deployment flexibility and reusability, while being able to combine existing data organization and processing components (i.e. libraries and computing frameworks) from different environments.

H. Anzt et al. (Eds.): ISC High Performance 2022 Workshops, LNCS 13387, pp. 221–232, 2022.
https://doi.org/10.1007/978-3-031-23220-6_15

HPC users typically integrate different stages of computation in custom scripts. In an effort to find a more expressive and portable language for defining application pipelines, we turned to existing work available in the "cloud-native" ecosystem. Bridging the two worlds is now possible as HPC installations increasingly support containers: reusable units of integrated, pre-packaged software that can run unmodified using different runtimes with minimal performance overheads. Singularity[1] [1] has become the *de facto* container runtime in HPC.

In Kubernetes [4] setups in the Cloud, Argo Workflows [2] is quickly gaining ground as the industry-standard workflow environment, providing a language and runtime to model and execute applications as directed acyclic graphs (DAGs). In Argo, every node of the graph is a container. The Argo controller processes each workflow by submitting respective containers for execution, monitoring their status, and collecting their outputs; all presented via a user-friendly, interactive, web-based frontend. The frontend also allows organizing workflows using templates, as well as planning repeated execution with a crontab-like syntax. Under the hood, Kubernetes delegates execution to available nodes. In this paper, we extend Kubernetes with a virtual node using *KNoC* (Kubernetes Node on HPC Cluster), which receives container management operations and forwards them to the cluster over ssh. KNoC consists of two main components: the virtual node at the Kubernetes side, which is implemented as a Virtual Kubelet [8] plug-in, and an executable (called *Door*) automatically installed at the HPC side, which provides a simple API for running the commands required to start and stop Singularity containers using Slurm [7]. KNoC is open-source [3].

In contrast to related works which bridge Cloud with HPC facilities through special "cluster job" objects in Kubernetes or other constructs, our approach allows us to transparently delegate execution to external compute clusters, while defining HPC-specific parameters directly in the workflow language. KNoC effectively elevates Argo workflows into a cross-platform standard, which can support open, reproducible science on heterogeneous computing facilities. With KNoC, Cloud users can easily tap on the vast amount of high-performance computing resources available in HPC centers, while HPC users can leverage the expressiveness and the interactivity of the Argo environment, while also exploiting the integration by mixing HPC workflow steps with other cloud-native utilities and runtimes. For instance, Argo Events enables triggering workflows on events coming from a variety of sources. The Kubernetes installation hosting the KNoC node may run locally to the user (in a simple virtual machine), or even in a "sidecar" environment offered by the HPC center.

2 Related Work

A tool for seamlessly combining jobs that run either in Kubernetes or in HPC environments is presented in [15]. *hpc-connector* acts as an HPC job proxy: users submit their jobs as hpc-connector instances with specific settings at the Kubernetes side, and hpc-connector forwards them to the HPC cluster, monitors their

[1] The Singularity project has recently been renamed to Apptainer.

execution, and collects their results. Docker and Singularity containers are also used to address portability issues. The authors identify five requirements for a system that allows running the same workloads on Cloud and HPC. Our work addresses all five, and extends the focus on workflow reproducibility and ease of use. We not only require the same *method* for job execution in both environments, but also the same *language* for defining workflows. With hpc-connector, users have to format their workloads to explicitly use it as a job-forwarding utility. Furthermore, since KNoC implements the functionality at the Kubernetes node level, it offers a generalized solution for forwarding container execution at the HPC side—not only for specific types of jobs. In [18], a Kubernetes installation is interfaced to a Torque-based HPC cluster, using a custom tool called Torque-Operator. Although this study offers the flexibility of running containerized Cloud and HPC jobs over the same front end interface through the WLM operator [9], it again uses a different language for describing jobs targeted for the HPC cluster.

Workflow reproducibility has recently been in the spotlight, especially in life sciences. Driven by the need to share both results and methods in a collaborative environment, as well as the need to verify the accuracy of computational results, a number of "infrastructure-agnostic" tools have been proposed. In the area of bioinformatics, examples include Nextflow [13], Snakemake [14] and Arvados [11]. These systems typically define their own DSLs (Domain Specific Languages) to construct workflows and most support containers for transparent job submission to either Cloud or HPC environments. As another example, the StreamFlow [12] runtime allows the execution of workflow steps onto multiple heterogeneous sites, automatically copying required data where needed. Groups of workflow steps may require specific environments to run, which are translated to runtime deployment dependencies. A proof-of-concept implementation uses the Common Workflow Language (CWL). In this paper we use the Argo language, however the hybrid Cloud-HPC platform offered by KNoC is language-independent and should be able to support other workflow runtimes as well. Also, it allows running workflows combining steps using other cloud-native frameworks, external to the workflow environment (within the limits described in Sect. 4.3).

3 Design

KNoC is implemented as a virtual node at the Kubernetes level—a virtual *kubelet*. The kubelet is the primary agent that runs on each node of a Kubernetes cluster. It is responsible for starting and stopping containers, reporting on their status, gathering their logs, etc. Each kubelet receives *PodSpecs*, which are Kubernetes objects that describe Pods and ensures that the containers comprising those Pods are running and healthy. Kubelets running on physical machines practically implement the interfacing between Kubernetes and the underlying container runtime; typically Docker or containerd. KNoC, on the other hand, directly manages containers that run on a remote HPC cluster. KNoC is implemented using Virtual Kubelet, an open source kubelet implementation featuring

a pluggable architecture for extensions that connect Kubernetes to other container execution environments. Virtual Kubelet provides the necessary features to support the lifecycle management of Pods (as a collection of containers) and supporting resources in the context of Kubernetes, while exposing simpler APIs at the back-end for plug-ins.

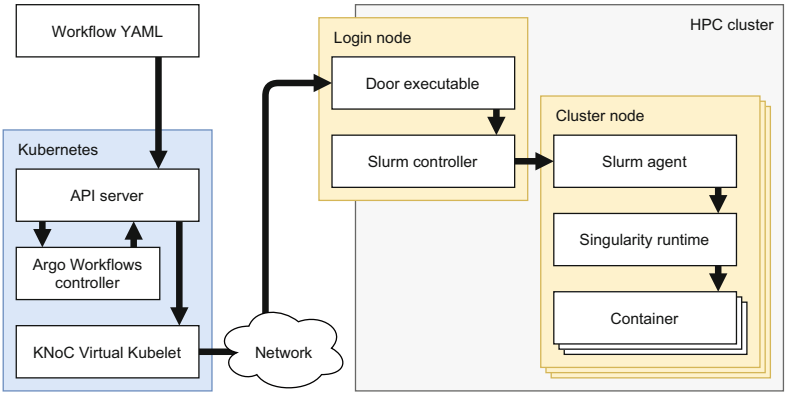

Fig. 1. Running a cloud-native workflow on an HPC cluster using KNoC

An overview of the main components involved in KNoC's deployment environment is shown in Fig. 1. The main goal is to run the container-based workflow provided by the user in Argo format (top-left) on the HPC cluster with Singularity containers (bottom-right). The Argo Workflows engine receives the workflow YAML and creates the corresponding Pods through the Kubernetes API. Then Kubernetes selects "KNoC" as the execution node to run the respective containers. If other, physical nodes also exist, this can be accomplished using node-selection constraints in the workflow language.

When KNoC receives a request to create a container as part of a Pod specification, it forwards the request to the remote HPC system supplied in the configuration. The assumption is that all interfacing with the remote environment can only happen by running commands through a secure shell (ssh) connection. To simplify the integration and add an abstraction layer at the remote end, the KNoC virtual kubelet installs and runs the Door agent remotely. Door receives simple requests from KNoC related to container creation and tear down, and produces the commands necessary to perform the respective actions. A temporary folder for each container at the HPC side keeps all runtime outputs and state, and is used by KNoC for monitoring the status of execution.

Door may implement different container execution plug-ins: it currently runs containers using Singularity, through Slurm, by creating sbatch scripts that form an execution environment for launching the containers. However, it can be easily reimplemented to use a different container runtime. Slurm is one of the most popular job schedulers used in HPC environments. It will distribute jobs—in our case

Singularity containers—across available resources. Singularity will automatically convert the Docker images used in the workflow to create Singularity-compatible image files that run with the same commands as their entrypoints. We use Singularity for container execution due to its wide-spread availability in HPC, thanks to its performance and security characteristics.

To access special hardware features of the HPC environment (such as GPUs), or specify other requirements or constraints at the level of the generated Slurm job (such as MPI parameters), the user can use specific labels in the workflow, which Door includes as flags in the respective Slurm command.

4 Implementation

4.1 The KNoC Virtual Kubelet Provider

KNoC is implemented as a Virtual Kubelet *provider*. Providers use Virtual Kubelet as a library which implements the core logic of a Kubernetes node agent (kubelet), and wire up their implementations for performing necessary actions. There are 3 main interfaces that a provider may offer:

- PodLifecycleHandler is consumed by the PodController which implements the core logic for managing Pods assigned to the node. Creating, updating, or deleting Pods in Kubernetes results in API calls for performing corresponding actions at the kubelet level.
- NodeProvider is responsible for notifying Virtual Kubelet about node status updates. Virtual Kubelet will periodically check the status of the node and update Kubernetes accordingly. The implementation of this interface is optional.
- PodNotifier is used by the provider to notify the Virtual Kubelet about Pod status changes. The implementation of this interface is optional.

The KNoC virtual kubelet implements the PodLifecycleHandler and PodNotifier interfaces. KNoC also introduces the RemoteExecutionHandler module that complements the PodLifecycleHandler, to handle the interaction with the remote execution environment. KNoC is written in Go, using approximately 1200 lines of code.

When a new Pod is created, KNoC's implementation of the PodLifecycleHandler will first go through the description and isolate any *initContainers*. A Pod can have multiple initContainers that need to run to completion sequentially before any other containers are started. Once the ordering of container execution is decided, KNoC will submit the containers in phases: first the initContainers, wait for them to complete, then the rest of the Pod members.

KNoC connects via ssh to the remote HPC cluster. The network address, username, and ssh key necessary to perform the connection are stored in a Kubernetes Secret and passed as environment variables to the KNoC executable on initialization. Also, on deployment, KNoC should be configured to advertise the total CPU cores and memory that are available at the cluster side, as Kubernetes keeps track of what resources have been allocated on each node.

For each container to be created, KNoC will:

1. Check if the Door binary is available remotely; if not, transfer it over.
2. Create a temporary folder in the form `~/.KNoC/<namespace>/<pod_uuid>/<container_name>/` for keeping files related to the execution of the respective container.
3. Place any attached Kubernetes Secrets and container environment variables as files in the temporary folder in key-value form.
4. Create folders for any attached Kubernetes *emptyDir* volumes in the temporary folder.
5. Run `Door submit` in the background, handing it over a JSON with details about the container and its environment, including references to the files and folders created above.

When the PodNotifier API implementation is triggered, KNoC starts a timer to periodically check the status of all Pods. The container execution command generated by Door places the containers' output and error streams, along with their exit codes into different files inside their temporary folders. KNoC uses the exit code files to monitor changes of container states, which are then consolidated to devise the corresponding Pod states that must be reported back to Kubernetes. To delete a Pod, KNoC calls `Door stop` remotely for each container. After the containers are stopped, the monitoring function will note the changes in exit codes and update their Pod status in Kubernetes.

4.2 The Door Executable

Our initial KNoC implementation produced a full Singularity command with environmental variables, mount paths, container commands, and arguments to be submitted over ssh. However, we decided to abstract this interaction into a simple API and synthesize the command at the remote end, using different implementations supporting different execution environments. This also solved the potential problem of having to run several commands (or a complete script) to manage container execution at the HPC cluster side. The remote functionality is realized by the Door executable, written in Go, using approximately 200 lines of code. Door started by using "plain" Singularity, while it currently uses Slurm to submit Singularity commands. Other Door implementations may use different container runtimes.

When Door is called to create a container:

1. It converts the given JSON description to an sbatch script that runs the respective Singularity command, including environment variables, values from Secrets, volumes, etc.
2. It submits the job to Slurm.
3. It writes down the resulting job id into a file in the container's temporary folder, which is used in case it needs to stop or cancel the job.

```
kind: Workflow
metadata:
  ...
spec:
  podMetadata:
    annotations:
      slurm-job.knoc.io/flags: "--mem=32gb"
      slurm-job.knoc.io/mpi-flags: "..."
  ...
```

Listing 1.1. Adding Slurm-specific annotations to workflows

Through Slurm, jobs may request exclusive or non-exclusive resources for execution. Door will pass particular annotations in the container description (coming from the respective Pod) to the generated Slurm command, so that the corresponding container will run with the specified resources. A simple example is shown in Listing 1.1, where the annotations used in the Workflow will be copied over by Argo Workflows to the Pods submitted for each step, and Door will use the value `--mem=32gb` verbatim when invocating the `sbatch` executable. If `mpi-flags` are defined, Door will invoke MPI to run the container with the additional parameters given. Also, container names may refer to either Docker images (which will automatically be converted to Singularity upon execution), or Singularity `.sif` files.

4.3 Integration with Argo Workflows

To successfully run Argo workflows with KNoC, we had to overcome several issues related to the availability of the Kubernetes API, volumes and associated data at the remote side. KNoC implements general-purpose remote container execution—amid, however, practical limitations. Local and remote systems may not share the same storage facilities. Also, the Kubernetes volume abstractions and mechanics are not available in the HPC environment.

For each workflow step, the Argo Workflows controller runs a sidecar container, called *executor*, in parallel to the "main" container defined by the user. The controller communicates with the executor to perform control actions (like kill, suspend, abort, etc.), monitor the state of execution, and collect outputs. Several executor implementations are available; each for a different container runtimes. When running Argo Workflows in Kubernetes, we select the "k8sapi" executor that actually uses the Kubernetes API to retrieve information and submit commands.

Running the Argo executor container remotely did not initially work. First, the executor could not communicate with the Kubernetes API. For this reason, we require that a `~/.kube/config` file is placed at the remote side, configured appropriately so that Kubernetes is accessible from the HPC cluster. This file is then available within containers, as Singularity automatically mounts the user's home folder in all containers. Most applications using Kubernetes API libraries will work without changes, as the libraries check for the file at predefined paths and use it. Future versions of KNoC will automatically create the Kubernetes configuration file inside the remote container's temporary folder, depending on the Namespace and ServiceAccount of the running Pod.

Second, the default Argo k8sapi executor (in version 3.0.2) uses the Kubernetes *Downward API* to examine the Pod's status. The Downward API is a method to provide Pod introspection in Kubernetes. When the Downward API is "mounted" within a Pod, all containers can access the Pod's status and annotations as files. Moreover, applications can monitor these files for changes to the Pod's state or configuration. In Argo Workflows 3.0.2, the controller mounts a Downward API volume at each executor instance, which is then used to get updates on the main container's execution status. Instead of implementing the Downward API functionality at the HPC cluster side, we changed the Argo controller and executor to not use it at all. This was already a request by the Argo Workflows community, as some Cloud providers do not support the Downward API in their Kubernetes nodes (i.e., in AKS virtual nodes). Our changes have been approved by the project's maintainers and the Downward API is no longer necessary.

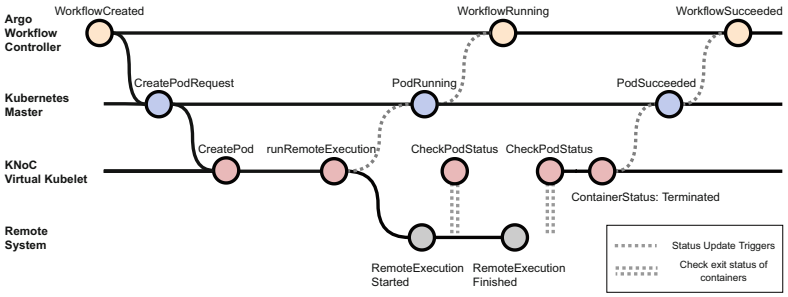

Fig. 2. Lifecycle of containers as part of an Argo workflow submitted to a remote system through KNoC

Figure 2 depicts the timeline of status updates, as workflow steps, corresponding Pods, and remote containers are created during the execution of an Argo workflow. A WorkflowCreated event happens when a workflow YAML is submitted to Kubernetes as a CRD instance. Then the Argo Workflows controller requests the creation of a number of Pods that relate to the workflow steps, by issuing respective CreatePodRequest calls at Kubernetes. Each call results in Kubernetes picking an appropriate Node to assign the Pod, and instructing that Node (through the kubelet API) to create the Pod including every container inside it. In the case of KNoC, once every container in the given Pod has been sent for remote execution, KNoC updates the Pod status to "Running". In turn, the workflow controller, which monitors Pod status changes, updates the workflow status to "Running". During this time, KNoC periodically polls for changes in remote container state. When the process running the container finishes, the exit code is written to a file. KNoC will pick up the change, and set the container status as "Terminated". If every container exits without errors, then the Pod status is updated to "Succeeded". As a result, the Argo controller marks the workflow step as "Succeeded" and moves on.

5 Evaluation

To evaluate KNoC, we deploy a minimal Kubernetes setup using minikube [5] in one machine, and configure KNoC to use the login node of our HPC cluster. As highlighted in Sect. 4.1, we instruct KNoC to report the sum of all cluster CPUs and memory to Kubernetes, so the latter will take full advantage of the available resources. We use Kubernetes version 1.19.10, Slurm 20.11.8, and Singularity 3.8.5. In our setup, we only use the KNoC node for scheduling pods, so all workflow steps will be sent to the cluster. In case KNoC runs alongside physical nodes, workflow specifications should be augmented with the appropriate NodeSelector, so their containers will be routed to the HPC side.

The KNoC source repository includes several workflow examples that we have used to evaluate the integration, including workflows defined as DAGs, loops, conditionals, etc. A simple example of an HPC workflow running the "embarassingly parallel" NAS benchmark [6] is shown in Listing 1.2. We use the language's `withItems` construct to spawn 4 parallel steps, each running another instance of the executable with different parameters. Also, note the use of the Slurm flag, defined as an annotation on the step template, to control the number of tasks used for each instance. This template showcases a method to run a parallel parameter sweep as part of a larger workflow. The "items" used may be explicitly set or be dynamically generated as the output of a previous step.

```
kind: Workflow
metadata:
  ...
spec:
  entrypoint: npb-with-mpi
  templates:
  - name: npb-with-mpi
    dag:
      tasks:
      - name: A
        template: npb
        arguments:
          parameters:
          - {name: cpus, value: "{{item}}"}
        withItems:
        - 2
        - 4
        - 8
        - 16
  - name: npb
    metadata:
      annotations:
        slurm-job.knoc.io/flags: "--ntasks={{inputs.parameters.cpus}}"
        slurm-job.knoc.io/mpi-flags: "..."
    inputs:
      parameters:
      - name: cpus
    container:
      image: mpi-npb:latest
      command: ["ep.A.{{inputs.parameters.cpus}}"]
```

Listing 1.2. A simple workflow executing parallel MPI steps

On the other hand, to better understand the issues involved in compiling workflows that can easily migrate from a cloud-native to a KNoC-based setup,

we use a real-life Argo workflow from the bioinformatics domain. This workflow performs genotype imputation [17], a computational method which is used to artificially increase the number of identified mutations in an input human DNA using a large dataset containing several thousand samples as a reference. The process, from a computational perspective, involves two basic steps: extracting the chromosomes from the input DNA and performing quality control/phasing, and then doing the actual imputation in batches of chromosomes, each measuring 5,000,000 base pairs long. The respective tools have been packaged into a container image, which is then used by the workflow. Each chromosome and each batch can be processed independently of each other, so each workflow phase deploys multiple containers in parallel, as shown in Fig. 3. The first phase processes 22 chromosomes and the second 589 ranges in parallel.

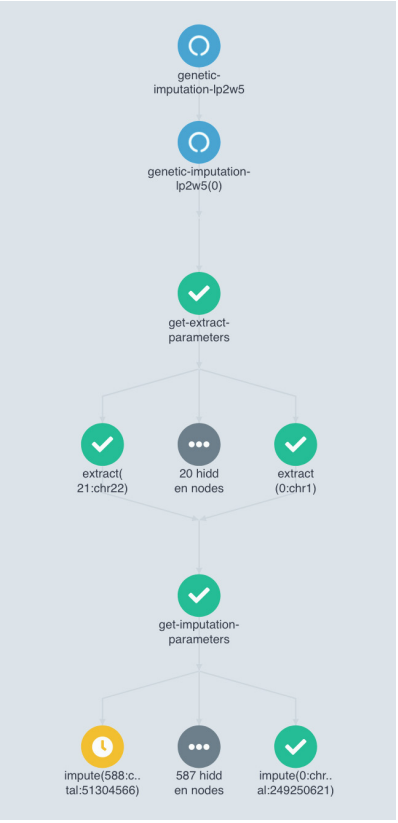

Fig. 3. The genotype imputation workflow as shown in Argo

A major point that should be considered when preparing cross-platform workflows (running on both Cloud and HPC sides) is *data availability*. Workflow

stages may require shared datasets or a mechanism for communicating processed data from one stage to the next. This can be achieved by using an Argo *artifact repository*, as a common place to deposit files, or a *shared folder*, mounted across all containers at a known path. Argo supports many S3-compatible services for artifacts; the executor will copy in specific files so they are available to running containers before startup and copy out results after stage completion. On the other hand, a shared folder has the benefit of avoiding data copies. Currently, for simplicity, we use USL [10] at the Kubernetes side, to provide all containers with a common mountpoint, and allow Door to use the default Singularity behavior of mounting the user's home folder at the cluster side. Then we define a workflow *parameter* that determines the base path that the workflow will use at runtime for data. In the imputation workflow, one preparatory step downloads the reference dataset in the shared folder (which is about 11 GB in size), and subsequent steps use it for writting out intermediate files and results. We plan to address data availability across environments in more detail in future work.

By using KNoC, we are able to easily scale out the workflow using the resources available at the HPC side. Argo provides a "parallelism" parameter to specify the maximum number of parallel pods that can run at the same time during execution, which in turn allows controling the maximum number of Slurm jobs that are submitted in parallel to the cluster. At the cluster side, job scheduling is exclusively handled by Slurm.

6 Conclusion

The distributed computing landscape is continuously growing with new Cloud and HPC offerings. Applications, expressed as workflows, deal with increasingly large and diverse datasets, requiring more and more processing capacity, as well as the integration of a variety of tools from both domains. The Cloud heavily relies on container-based technologies to provide standardization across providers and portability of execution. With the same abstractions available at "traditional" HPC installations, we can now embrace the heterogeneity of available platforms under a common higher-level workflow language and enable workloads to exploit all available resources. KNoC is a step in the direction of bridging Cloud and HPC computing. It adds a virtual node at the Kubernetes layer, which acts as a proxy orchestrating container execution at the HPC cluster using Slurm and Singularity. KNoC allows *any* Kubernetes Pod to run remotely—not just workflow steps. In this paper, we present the design and implementation of KNoC, and focus on its applicability from both the HPC and Cloud perspectives, by examining the integration of Argo Workflows within the KNoC-based system and discussing on the issues that must be considered when constructing applications.

Acknowledgements. We thankfully acknowledge the support of the European Commission under the Horizon 2020 Programme through project HiPEAC (GA-871174), as well as the European Commission and the Greek General Secretariat for Research and Innovation under the EuroHPC Programme through projects EUROCC (GA-951732) and DEEP-SEA (GA-955606). National contributions from the involved state members (including the Greek General Secretariat for Research and Innovation) match the EuroHPC funding.

References

1. Apptainer. https://apptainer.org
2. Argo workflows. https://argoproj.github.io/projects/argo
3. Knoc: A kubernetes node to manage the container lifecycle on an hpc cluster. https://github.com/CARV-ICS-FORTH/KNoC
4. Kubernetes: Production-grade container orchestration. https://kubernetes.io
5. Minikube. https://minikube.sigs.k8s.io
6. Nas parallel benchmarks. https://www.nas.nasa.gov/software/npb.html
7. Slurm workload manager. https://slurm.schedmd.com/documentation.html
8. Virtual-kubelet. https://github.com/virtual-kubelet/virtual-kubelet
9. Wlm-operator. https://github.com/sylabs/wlm-operator
10. Chazapis, A., Pinto, C., Gkoufas, Y., Kozanitis, C., Bilas, A.: A unified storage layer for supporting distributed workflows in kubernetes. In: Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems. CHEOPS 2021 (2021)
11. Chojnacki, S., Cowley, A., Lee, J., Foix, A., Lopez, R.: Programmatic access to bioinformatics tools from embl-ebi update: 2017. Nucleic Acids Res. **45**(W1), W550–W553 (2017)
12. Colonnelli, I., Cantalupo, B., Merelli, I., Aldinucci, M.: Streamflow: cross-breeding cloud with hpc. IEEE Trans. Emerg. Topics Comput. **9**(04), 1723–1737 (2021)
13. Di Tommaso, P., Chatzou, M., Floden, E.W., Barja, P.P., Palumbo, E., Notredame, C.: Nextflow enables reproducible computational workflows. Nat. Biotechnol. **35**(4), 316–319 (2017)
14. Köster, J., Rahmann, S.: Snakemake-a scalable bioinformatics workflow engine. Bioinformatics **28**(19), 2520–2522 (2012)
15. López-Huguet, S., Segrelles, J.D., Kasztelnik, M., Bubak, M., Blanquer, I.: Seamlessly managing HPC workloads through kubernetes. In: Jagode, H., Anzt, H., Juckeland, G., Ltaief, H. (eds.) ISC High Performance 2020. LNCS, vol. 12321, pp. 310–320. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59851-8_20
16. Ungerer, T., Carpenter, P., et al.: Eurolab4HPC Long-Term Vision on High-Performance Computing, 2nd edn. (2020). https://www.eurolab4hpc.eu/media/public/vision/vision_final.pdf
17. Van Leeuwen, E.M., et al.: Population-specific genotype imputations using minimac or impute2. Nat. Prot. **10**(9), 1285–1296 (2015)
18. Zhou, N., Georgiou, Y., Zhong, L., Zhou, H., Pospieszny, M.: Container orchestration on hpc systems. In: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), pp. 34–36 (2020)