Running Kubernetes Workloads on HPC

Antony Chazapis[1(✉)], Fotis Nikolaidis[1], Manolis Marazakis[1], and Angelos Bilas[1,2]

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

Abstract. Cloud and HPC increasingly converge in hardware platform capabilities and specifications, nevertheless still largely differ in the software stack and how it manages available resources. The HPC world typically favors Slurm for job scheduling, whereas Cloud deployments rely on Kubernetes to orchestrate container instances across nodes. Running hybrid workloads is possible by using bridging mechanisms that submit jobs from one environment to the other. However, such solutions require costly data movements, while operating within the constraints set by each setup's network and access policies. In this work, we explore a design that enables running unmodified Kubernetes workloads directly on HPC. With *High-Performance Kubernetes* (HPK), users deploy their own private Kubernetes "mini Clouds", which internally convert container lifecycle management commands to use the system-level Slurm installation for scheduling and Singularity/Apptainer as the container runtime. We consider this approach to be practical for deployment in HPC centers, as it requires minimal pre-configuration and retains existing resource management and accounting policies. HPK provides users with an effective way to utilize resources by a combination of well-known tools, APIs, and more interactive and user-friendly interfaces as is common practice in the Cloud domain, as well as seamlessly combine Cloud-native tools with HPC jobs in converged, containerized workflows.

Keywords: Cloud-HPC convergence · Kubernetes · Virtual Kubelet · Slurm · Singularity · Apptainer

1 Introduction

Both Cloud and High-Performance Computing (HPC) setups offer developers computing environments to deploy large-scale applications, each with its unique development tools and supporting utilities. The choice of platform usually depends on the design characteristics and architecture of the application, or requirements applying to the software frameworks utilized. As an example, it is common to use HPC for running tightly parallelized codes performing large simulations, while the Cloud is a better fit for deploying out elastic webs of microservices or Big Data runtimes. This dichotomy is challenged by the increasing complexity and diversity of large workloads that tend to be composed of

A. Bienz et al. (Eds.): ISC High Performance 2023 Workshops, LNCS 13999, pp. 181–192, 2023.
https://doi.org/10.1007/978-3-031-40843-4_14

multiple processing stages in the form of workflows. *Convergence* is essential for developers of big processing pipelines, as they would like to effortlessly combine Cloud with HPC steps and seamlessly transition between execution environments, using the most effective and efficient solution for each step.

Up to now, Cloud-HPC convergence has generally been realized with interfacing mechanisms for submitting HPC jobs from the Cloud side or vice versa. However, bridging separate Cloud and HPC installations has several disadvantages, as it requires synchronizing data between sites, each with its own storage, data transfer, and authorization restrictions. Having two separate setups also elevates the associated hardware and maintenance costs.

To this end, we explore an HPC-centric solution that accommodates both Cloud and HPC on the same hardware. We focus our work on Kubernetes [9], the most popular distributed container orchestrator in the Cloud [22] and the runtime of choice for supporting the "Cloud-native" ecosystem [4]. We present the design and implementation of *High-Performance Kubernetes* (HPK), an open source integration of unmodified Kubernetes components and custom modules that runs as a user-level service, which in turn acts as a translator from Kubernetes-native descriptions of services and jobs, to Slurm [16] and Singularity/Apptainer [1,17,23] scripts that run on a typical HPC cluster. By delegating execution to Slurm, HPK "mini Clouds" comply with organization policies and established resource accounting mechanisms. HPK requires minimal support from the HPC environment, all being changes to the container runtime configuration, in order to enable private, inter-container communication across cluster nodes and the ability to start containers that internally run commands as arbitrary users.

HPK successfully runs several Cloud-native frameworks without modifications. This includes Argo Workflows [2] with placement of artifacts on MinIO [10] (an S3 service), as well as examples using the Spark operator [8] and TensorFlow Serving [18]. We expect this technology to play a significant role in the world of HPC, as it enables existing HPC users to tap on the vast collection of available Cloud applications and services, as stand-alone solutions or in hybrid computation scenarios combining Cloud frameworks with HPC codes. HPC centers may no longer need to maintain separate hardware partitions for Cloud analytics and HPC, as HPK allows running the Cloud workloads on the main HPC partition. Furthermore, HPK can be used to attract Cloud users to large HPC installations, offering them a familiar interface to seamlessly exploit the raw computing power available.

2 Related Work

We classify work related to Cloud-HPC convergence in two main categories: Systems that maintain the *separation* of Cloud and HPC resources and systems that *embed* one resource management framework into the other. In the former case there are two separate resource managers, while in the latter there is a single authority that controls hardware allocations, shared by both Cloud and HPC

deployments; the embedded framework delegates resource management decisions to the overall cluster manager. Works that assume separate clusters can further be divided into *bridging* solutions that operate within the context of the Cloud or HPC runtime, allowing the transparent submission of remote jobs, or *third-party* systems that operate in their own context and are able to administer tasks in both remote Cloud and HPC installations.

Many bridging solutions are available for Kubernetes, enabling Cloud users to integrate the execution of remote HPC jobs into their workflows. In [24], the authors use a utility called *hpc-connector* that acts as an HPC job proxy: Users submit Kubernetes jobs with specific settings, and the hpc-connector forwards them to the HPC cluster, monitors their execution, and collects their results. In [29], a Kubernetes installation is interfaced to a Torque-based HPC cluster, using a custom tool called *Torque-Operator*. The *Bridge Operator* [25] has similar goals and a wider compatibility of remote job execution facilities. All aforementioned projects use language extensions (Kubernetes custom resources) for describing jobs targeted for the HPC cluster. On the other hand, KNoC (Kubernetes Node on HPC Cluster) [26] is a virtual node for Kubernetes that transparently manages the container lifecycle on a remote HPC cluster using Slurm and Singularity. This technique effectively allows users to employ existing Cloud-native tools, such as Argo Workflows to express complex data-processing pipelines for both Cloud and HPC without explicit remote execution steps.

Bridging solutions are especially useful when Cloud and HPC resources are colocated. HPC centers increasingly support on-demand provisioning of Cloud resources—even as partitions of the main HPC machine [14]. However, when the two are remotely situated, bridging suffers from the overhead of maintaining data copies. The user must prepare and send inputs to the remote HPC cluster before issuing any tasks, and then place back outputs in the Kubernetes context. Data synchronization in hybrid workflows is addressed by StreamFlow [21], a third-party system, which extends the workflow language with declarative descriptions of execution sites (either Cloud or HPC) and their relationship to workflow nodes. The runtime automatically infers data dependencies, so to copy required data where needed before running each step.

Embedded convergence solutions avoid data copies and the requirement to manage and maintain two separate setups, as both Cloud and HPC share a common hardware platform. In [28], the authors embed the HPC runtime in Kubernetes, by introducing the concept of the *virtual cluster*, as a group of multiple container instances that function as a private HPC cluster for a user (similar to [11]). Each node in a virtual cluster includes all necessary libraries and utilities, as well as a full Slurm deployment; the user working inside a virtual cluster can only view and manage jobs submitted from within the same context. To coordinate resource allocations between tasks running within virtual clusters and other Kubernetes services, the Slurm controller is extended with a custom protocol that requests resources from the Kubernetes scheduler, effectively placing Kubernetes in charge of resource management for the whole cluster. A custom Kubernetes scheduler is also employed, in order to apply differ-

ent container placement policies for "HPC" and "data center" services (typical Kubernetes deployments that run in other containers).

From Slurm's perspective, several embedded configurations are presented in [27]: *Over* is defined as the setup where Slurm is in control of the cluster, creating Kubernetes environments ephemerally within batch jobs, *adjancent* when both Slurm and Kubernetes are installed on the same physical nodes but share a common scheduler (i.e., Kubernetes uses Slurm to place jobs [15]), and *under* when Slurm-enabled pods are deployed in Kubernetes (like *virtual clusters*). HPK falls within the *over* class, however it does not create a full Kubernetes environment as a batch job, but rather transforms each Kubernetes-level deployment as an individual Slurm script, allowing for better scheduling flexibility and finer grain resource sharing.

We are not aware of any other system that embeds Kubernetes in HPC. Usernetes [19] is a step in this direction, providing a Kubernetes distribution that can run without root privileges. We did consider extending Usernetes to implement HPK, but quickly realized that the necessity of interfacing with Slurm and Singularity/Apptainer at multiple levels, would require reevaluating the internal structure of Kubernetes leading to reimplementing several subsystems. Interestingly, Usernetes solves the problem of managing the system's routing tables by utilizing a user-level networking stack, although this imposes several requirements to the environment, including availability of specific kernel modules.

3 Design

Our goal is to provide a mechanism for HPC users to run Kubernetes workloads in a typical cluster environment, so they can easily deploy hybrid workflows that combine Cloud-native frameworks with MPI codes. From a design perspective, the requirements for this mechanism are:

- Compatibility: All Kubernetes abstractions should be available and fully functional, except those that directly relate to physical hardware resources (i.e., "NodePort" services that request a specific port number for exposing services at Kubernetes nodes). Higher-level constructs, such as pods (one or more containers that are scheduled and scaled as a group), deployments, services, jobs, and volumes should be applicable with no changes. For supporting microservices, pods should be individually addressable, with inter-container networking and internal service discovery working as expected.
- Compliance: All resource management decisions should be delegated to the cluster manager operating the cluster (i.e., Slurm). Organization policies for resource allocation and accounting should be fully respected. Forwarded workloads should include as much information as possible, so the cluster manager can know at any point what is actually running. Also, minimal (ideally none) configuration changes should be required to be done at the host level by HPC administrators. Reliance on special libraries or binaries that execute with "elevated" permissions should be avoided.

– Usability: Make it easy for end users. Provide one simple command or script to deploy. All binaries should be neatly packaged up with their dependencies into a container, with no host-specific requirements. All relevant configuration and files should reside in the user's home directory.

To this end, we started by envisioning HPK as a user process; a Kubernetes-in-a-box integration, packaging the official Kubernetes binaries in a container that the user would be able to launch using a simple Slurm script. Kubernetes is implemented as a set of communicating subsystems that collectively provide the functionality of distributed container orchestration. A typical Kubernetes deployment constitutes of the following (Fig. 1):

– API server: The "heart" of Kubernetes. The main interface to the cluster and the synchronization point for all controllers.
– etcd: The key-value store holding all state. Always accessed through the API server.
– Controller manager: Watches for configuration changes or failures and performs all necessary actions to reach the desired state set by the user. The controller manager includes the controllers that implement the logic for the base Kubernetes abstractions. As all controllers, it communicates only with the API server.
– Scheduler: A controller that decides which node will be used to run new pods.
– CoreDNS: A controller and DNS server for implementing naming and discovery for pods and internal cluster services.
– Kubelet: An agent running on each worker node, implementing the pod lifecycle using a specific container runtime (i.e., Docker or containerd directly).
– Network plugin: A service supporting the Container Network Interface (CNI) specification that assigns addresses to pods. The network plugin, which—depending on the Kubernetes version—is used by the kubelet or the container runtime directly, implements the Kubernetes network model [7]. In addition to assigning unique, cluster-wide addresses, it makes sure that pods can communicate with each other across hosts. It may also realize traffic shaping policies or other network-level features.
– Proxy: Creates local network routes for virtual IP addresses used by services (i.e., for load balancing). Runs on each worker node, alongside the kubelet.
– Storage controller: A controller that provisions storage of some type that can be attached to pods. Creates physical volumes to match requested persistent volume claims. While this controller is optional, we consider it a core component for a functioning system.

From these, we expected that the *API server*, *etcd*, *controller manager*, and *CoreDNS* would require no changes, as they implement specific functions that do not interface with the execution environment. On the other hand, assuming that most HPC centers include Singularity/Apptainer as part of their standard software environment, we would certainly require a custom *kubelet* to interface with the container runtime. We considered several options on how to layout the cross-node Kubernetes components; whether to keep the arrangement of separate

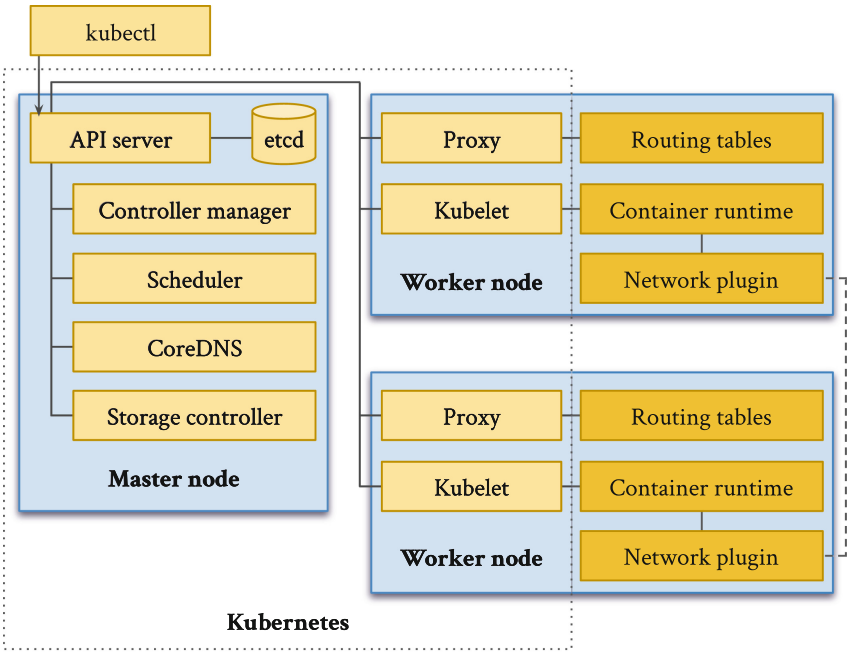

Fig. 1. Components involved in a typical Kubernetes deployment on bare-metal

Kubernetes agents running on every node, or use a single kubelet representing the whole HPC cluster as one execution entity.

The former would require preallocating resources in several machines for a multi-node deployment and then managing them from Kubernetes—essentially running Kubernetes as one large Slurm job, spanning multiple nodes, which would then internally schedule and place its own workloads. This solution, however, had several shortcomings. First, it would result in large, coarse-grain allocations, that would then need to be filled up with jobs, leading to less flexibility (resource allocations in Slurm are static). Second, although not violating the *compliance* requirement, the actual Kubernetes workloads would not show up in Slurm, but remain hidden underneath the overall Kubernetes job. Only Kubernetes would know the actual characteristics of each embedded workload, including its size and duration.

For these reasons, it seemed reasonable to use a single, virtual kubelet in HPK, which instead of interfacing directly with the container runtime, would layer above both *Slurm and Singularity/Apptainer* for execution. In practice, the *hpk-kubelet* is a translator from Kubernetes semantics to Slurm scripts, as shown in Fig. 2. With a single kubelet proxying requests to Slurm, the whole HPK integration can be visualized as a *translation service*: Workloads enter in YAML format through the Kubernetes API endpoint and exit as Slurm scripts from hpk-kubelet. They transparently show up in Slurm queues, and their Kubernetes-level resource requirements end up as allocation requests to Slurm. This architecture has no special requirements for HPK as a whole; it can all run with minimal resources on any cluster node. Additionally, with a single kubelet, the *scheduler* can be greatly simplified. Since cluster-level scheduling

is performed by Slurm, the HPK scheduler should always select hpk-kubelet, regardless of actual resource availability.

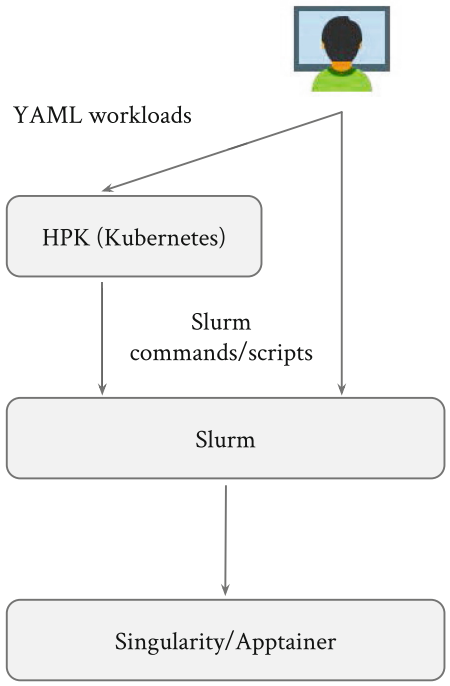

Fig. 2. HPK translates Kubernetes workloads to Slurm and Singularity/Apptainer

The container runtime used also directly influences the mechanisms implementing virtual addresses and networks employed internally by Kubernetes. In a bare-metal Kubernetes setup, there are actually three networks involved: The physical network between hosts, the—typically virtual—network used by pods, and the virtual network used by services. While each uses a different IP address range, the *network plugin* maintains the necessary routes for cross-network communication between pods (and pods with hosts), while the *proxy* manages the respective rules for services. HPK could not include any of these subsystems, as they perform actions at the system level as the root user. Our approach for pod addresses was to require that a corresponding network plugin is configured at the Singularity/Apptainer level by the HPC administrators. Singularity/Apptainer supports CNI plugins and can be easily set up to delegate network addressing to a cluster-wide service (i.e., Flannel [5]). This, in addition to allowing containers to run as *fakeroot* for supporting common Docker images that use the root user, are the only changes HPK requires from the HPC environment; both being configuration options of the container runtime.

The service-level network is used by "ClusterIP" services. When such a service is created, the Kubernetes control plane assigns a new virtual IP and the *proxy* adds respective rules at the host to redirect traffic to a pod that implements the service, or to load-balance between available backend pods. Supporting this functionality without being able to manipulate the routing tables of the host is

impossible, so we chose to completely disable "ClusterIP" services, making the proxy obsolete. This would be possible via a Kubernetes admission controller—a hook that monitors API requests and may reject or mutate them before reaching the API server. In Kubernetes, services can explicitly request to not use a virtual service IP (also called "headless" services). In such cases, service discovery continues to function, as CoreDNS maps the service name to the actual pod IPs instead of the virtual service address. Thus, microservice architectures are not affected by the lack of service-specific IPs. If load-balancing between pods is necessary, it can be implemented by using an additional service within a deployment.

For storage, we experimented with various existing offerings and found that it would be straight-forward to integrate a simple *storage controller* for binding directories inside the user's home to containers as volumes (i.e., similar to "HostPath" volumes).

The overall architecture of HPK is shown in Fig. 3.

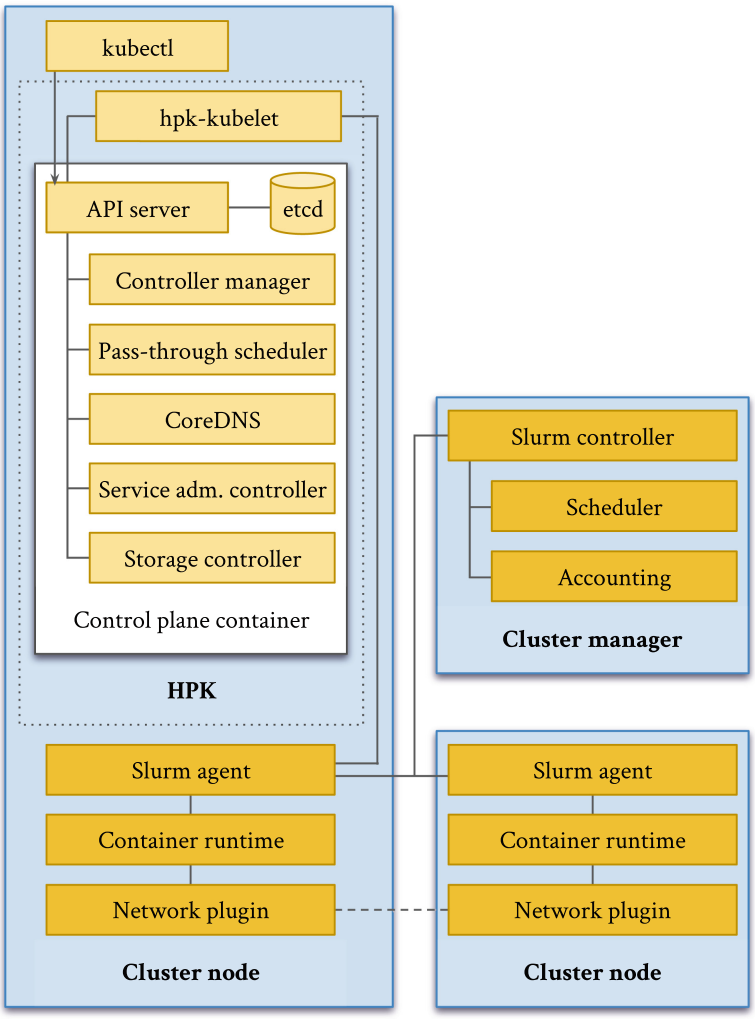

Fig. 3. HPK architecture

4 Implementation

HPK implementation started by integrating most Kubernetes submodules and services into a container. The resulting *Kubernetes-from-scratch* image recipe downloads and builds all relevant binaries. At runtime, necessary keys and certificates are generated, and the control plane is bootstrapped by initializing and starting the executables in order. These include:

- Unmodified versions of the *API server*, *etcd*, *controller manager*, and *CoreDNS*. As expected, these required no changes from the official releases.
- A simple *scheduler* that just selects the first available node.
- A simple admission controller that converts all services to "headless", by explicitly setting the "ClusterIP" to "None".

We start this container using Singularity/Apptainer and wait for it to produce the configuration file containing the endpoint and credentials needed to connect to the API server. This "control plane" container uses a virtual, private IP, which is accessible by all hosts in the cluster, as the CNI plugin enabled in Singularity/Apptainer has set up the appropriate network routes. Then, we start *hpk-kubelet*, which uses the configuration file to connect to the API server and announce its availability as a node. Without hpk-kubelet, pods can be created in the API server without ever transitioning to a running state.

Our custom *kubelet* is implemented as a *Virtual Kubelet Provider*. Virtual Kubelet [20] is an open source project that offers an intermediate, simpler API to implement a kubelet; it is mostly used to easily submit containerized jobs for execution on serverless platforms. Providers use Virtual Kubelet as a library which implements the core logic of a node agent, and wire up their specific implementations for supporting the lifecycle of pods and supporting resources on non-standard container execution environments. The hpk-kubelet translates Kubernetes actions into Slurm scripts using Singularity/Apptainer commands.

The main challenge faced when implementing hpk-kubelet, was respecting the network semantics of pods. In Kubernetes, each pod may include multiple containers in the same network namespace. Containers within the same pod share the same external IP address and can use localhost (IP address 127.0.0.1) to communicate with each other internally. As Singularity/Apptainer does not support attaching a container to an existing namespace, we had to produce an embedded container topology: hpk-kubelet starts a "parent" container, which in turn uses Singularity/Apptainer to run each pod container. The pod IP address is assigned to the parent container; "child" containers run within the same network context without extra IP addresses. Synchronization between the hpk-kubelet and parent containers is achieved through files placed under `~/.hpk`. The `~/.hpk` directory, which is used for holding the state of all running Slurm jobs and respective containers, is mounted by default in all parent containers.

In hpk-kubelet, creating a new pod is implemented as follows:

1. A request for a new pod is received.
2. A pod-specific directory is created and listeners are established to watch for changes in included files.

3. Two scripts are generated: A Slurm script that starts the parent container and a secondary script that runs within the parent and starts all child containers. The secondary script also includes all necessary environment variables expected to be set in the Kubernetes context (mainly information on available services and their respective ports) and volume mounts. The resulting job identifier is recorded in a file within the pod directory.
4. Slurm starts the parent container, which in turn starts its children. The pod directory is used by both scripts to save the generated IP address, as well as exit codes and output of all containers.
5. The listener may react to changes on these files to calculate the new state of the pod (i.e., a change in exit codes may trigger marking the pod as "completed", either with a success or failure).
6. The control plane is informed about the new state.

The hpk-kubelet will convert the requested allocation at the Kubernetes level to corresponding flags specified in the preamble of the Slurm script. Additionally, similar to KNoC [26], it will pass-through specific pod annotations unmodified as additional flags, so that the user may further customize execution. This also allows running a single container as a job in Kubernetes and scaling it up in the HPC environment via MPI-specific Slurm parameters. The usage pattern is unusual for Cloud users, but may prove helpful in HPC, as it allows compiling workflows where each step's scalability is individually controlled using just the annotations. A simple example of an Argo workflow with an HPC running the "embarassingly parallel" NAS benchmark [12] is shown in Listing 1. We use the language's `withItems` construct to spawn 4 parallel steps, each running another instance of the executable with different parameters. Note the use of a 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.hpk.io/flags: "--ntasks={{inputs.parameters.cpus}}"
        slurm-job.hpk.io/mpi-flags: "..."
```

```
25      inputs:
26        parameters:
27        - name: cpus
28      container:
29        image: mpi-npb:latest
30        command: ["ep.A.{{inputs.parameters.cpus}}"]
```

Listing 1. A simple Argo workflow executing multiple MPI steps in parallel, each with a different number of Slurm tasks

Deleting a pod results in canceling the respective Slurm job, which in turn updates the exit code of the parent container script that triggers cleanup. The latter requires updating the API server and removing hpk-kubelet state, including the pod directory.

For storage provisioning, we have currently integrated OpenEBS [13] in HPK and configured it to create directories under ∼`/.hpk` to match volume claims.

HPK is open source and available online [6]. The repository includes documentation and scripts to deploy test environments in AWS ParallelCluster [3]. Most significant open development tasks include mapping GPU/accelerator requests from Kubernetes to Slurm and handling port forwarding from services to the cluster's login node.

5 Conclusion

Kubernetes has become the industry standard runtime in the Cloud, providing the necessary abstractions to embrace the breadth and heterogeneity of available resources. Compatible Cloud-native tools are constantly evolving, covering a wide spectrum of applications, including database and queuing systems, interactive code execution frontends, workflow management utilities, as well as development frameworks that automatically optimize and scale operations. The ability to deploy this software in an HPC cluster via *High-Performance Kubernetes* (HPK) opens up new possibilities for both Cloud and HPC users.

HPK simply runs as a user-triggered service. Container workloads are handled by the hpk-kubelet executable—a virtual Kubernetes node representing the entire HPC cluster as a single entity. The hpk-kubelet translates container lifecycle actions to Slurm scripts and commands that internally use Singularity/Apptainer. HPK also includes several other customized Kubernetes modules to facilitate integration with the HPC environment and simplify adoption by HPC centers.

Acknowledgement. We thankfully acknowledge the support of the European Commission and the Greek General Secretariat for Research and Innovation under the EuroHPC Programme through projects EUPEX (GA-101033975) 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. AWS ParallelCluster. https://aws.amazon.com/hpc/parallelcluster/
4. Cloud native computing foundation. https://www.cncf.io
5. Flannel. https://github.com/flannel-io/flannel
6. High-performance kubernetes. https://github.com/CARV-ICS-FORTH/HPK
7. The kubernetes network model. https://kubernetes.io/docs/concepts/services-networking/#the-kubernetes-network-model
8. Kubernetes operator for apache spark. https://github.com/GoogleCloudPlatform/spark-on-k8s-operator
9. Kubernetes: Production-grade container orchestration. https://kubernetes.io
10. Minio. https://min.io/
11. MPI operator. https://github.com/kubeflow/mpi-operator
12. NAS parallel benchmarks. https://www.nas.nasa.gov/software/npb.html
13. OpenEBS: Kubernetes storage simplified. https://openebs.io/
14. S8s: Slurmenetes managed kubernetes service on meluxina HPC. https://jpclipffel.s3.lxp.lu/userdoc/cloud/s8s/index.html
15. slurm-k8s-bridge: Experimental slurm scheduling plugin for kubernetes. https://gitlab.com/SchedMD/training/slurm-k8s-bridge
16. Slurm workload manager. https://slurm.schedmd.com/documentation.html
17. Sylabs: Singularity container technology & services. https://sylabs.io
18. TensorFlow serving. https://github.com/tensorflow/serving
19. Usernetes: Kubernetes without the root privileges. https://github.com/rootless-containers/usernetes
20. Virtual-kubelet. https://github.com/virtual-kubelet/virtual-kubelet
21. Colonnelli, I., Cantalupo, B., Merelli, I., Aldinucci, M.: StreamFlow: cross-breeding cloud with HPC. IEEE Trans. Emerg. Top. Comput. **9**(04), 1723–1737 (2021)
22. Coté, M.: Kubernetes is here to stay: this is why (2022). https://tanzu.vmware.com/content/blog/state-of-kubernetes-2022
23. Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: scientific containers for mobility of compute. PLOS ONE **12**(5), 1–20 (2017)
24. 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
25. Lublinsky, B., Jennings, E., Spišaková, V.: A kubernetes 'bridge' operator between cloud and external resources (2022). https://arxiv.org/abs/2207.02531v1
26. Maliaroudakis, E., Chazapis, A., Kanterakis, A., Marazakis, M., Bilas, A.: Interactive, cloud-native workflows on HPC using KNoC. In: Anzt, H., Bienz, A., Luszczek, P., Baboulin, M. (eds.) ISC High Performance 2022. LNCS, vol. 13387, pp. 221–232. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-23220-6_15
27. Wickberg, T.: Slurm and/or/vs kubernetes (2022). https://slurm.schedmd.com/SC22/Slurm-and-or-vs-Kubernetes.pdf
28. Zervas, G., Chazapis, A., Sfakianakis, Y., Kozanitis, C., Bilas, A.: Virtual clusters: isolated, containerized HPC environments in kubernetes. In: Anzt, H., Bienz, A., Luszczek, P., Baboulin, M. (eds.) ISC High Performance 2022. Lecture Notes in Computer Science, vol. 13387, pp. 347–357. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-23220-6_24
29. 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)