Leveraging private container networks for increased user isolation and flexibility on HPC clusters

Lise Jolicoeur, François Diakhaté, Raymond Namyst

Leveraging private container networks for increased user isolation and flexibility on HPC clusters

Lise Jolicoeur (✉)[1,2], François Diakhaté[1], and Raymond Namyst[2][0000−0001−7734−1258]

[1] CEA, DAM, DIF, Arpajon 91297, France
firstname.lastname@cea.fr
[2] University of Bordeaux, INRIA, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France
raymond.namyst@u-bordeaux.fr

Abstract. To address the increasing complexity of modern scientific computing workflows, HPC clusters must be able to accommodate a wider range of workloads without compromising their efficiency in processing batches of highly parallel jobs. Cloud computing providers have a long history of leveraging all forms of virtualization to let their clients easily and securely deploy complex distributed applications and similar capabilities are now expected from HPC facilities.
In recent years, containers have been progressively adopted by HPC practitioners to facilitate the installation of applications along with their software dependencies. However little attention has been given to the use of containers with virtualized networks to securely orchestrate distributed applications on HPC resources.
In this article, we describe a way to leverage network virtualization to benefit from the flexibility and isolation typically found in a cloud environment while being as transparent and as easy to use as possible for people familiar with HPC clusters. Users are automatically isolated in their own private network which prevents unwanted network accesses and allows them to easily define network addresses so that components of a distributed workflow can reliably reach each other. We describe the implementation of this approach in the pcocc (private cloud on a compute cluster) container runtime. We evaluate both its overhead as well as its benefits for representative use-cases on a Slurm based cluster.

Keywords: HPC · containers · cloud · VXLAN · Slurm · namespaces · PCOCC

1 Introduction

In the last decade, high performance computing has been adopted in increasingly diverse scientific domains, from climate modelling and high energy physics to genomics and AI. HPC clusters have historically focused on providing platforms optimized for compute-intensive numerical simulations, which harness the

aggregate performance of a large number of nodes to perform computations in a minimal time on a fixed amount of resources using a common set of parallel programming interfaces such as MPI and OpenMP. HPC facilities have dedicated a large part of their efforts to serving this use case by providing a uniform, highly tuned software stack made of compilers, runtime systems, and libraries that users can leverage to build optimized parallel applications. However, with the increasing adoption of HPC across all scientific domains, new applications need to be able to benefit from HPC resources. As a global trend, they require a wider range of software dependencies and rely on more and more complex workflows mixing numerical simulations, AI/ML algorithms or data analysis. It is therefore becoming essential that HPC clusters provide tools that make it easier for users to deploy their own customized software stack. In contrast, cloud environments have been built from the ground up with the objective to securely run complex dynamic workloads while being shared by a very large pool of untrusted users. By providing flexibility, automation, elasticity, and scalability, cloud computing has become mainstream regarding the deployment of large-scale applications and workflows. These applications are often designed as a set of components or services, each requiring a specific runtime or software stack. Cloud providers make extensive use of hardware virtualization and containers to partition resources in a flexible and secure way as well as to facilitate application deployment by allowing the packaging of application components as shippable artifacts. The term *cloud-native* has been coined to describe applications that have been developed specifically for cloud platforms. The most popular platform for deploying these applications is Kubernetes[1], an open-source product that allows users to easily and reproducibly define and deploy their workloads on any kind of supported resource from commercial cloud offerings to self-hosted clusters or even edge devices. Kubernetes mainly supports orchestrating applications packaged as containers, a form of lightweight virtualization that operates at the level of operating system interfaces. In Linux, this capability is exposed in the form of namespaces, which allow changing the mapping of system identifiers such as PIDs, filenames, or network interfaces for a process or group of processes. By providing each application with its own virtual view of the operating system, applications can be isolated from each other and remain independent of the host system. While Kubernetes was initially primarily used to host web services, its flexibility has made it popular for running various workloads. Workflow engines enable the scheduling of complex workflows involving compute-intensive tasks on Kubernetes clusters, particularly in the AI/ML field.

While cloud and HPC have long evolved separately, there is a growing interest in bridging the gap between these two environments so that users may benefit from the best capabilities of both cloud and HPC technologies, an approach often referred to as "converged computing". In this article, we discuss how containers can help securely enable new workflows in typical HPC clusters in particular by making use of network virtualization, which is typically used in cloud settings. Indeed while HPC applications are now more and more commonly packaged in container images that can be reproducibly deployed, little attention

has been given to leveraging network virtualization within an HPC environment. To our knowledge, most HPC oriented container engines, with the exception of Singularity, do not support setting up network namespaces, and little work has been published on how to expose this capability to users of an HPC cluster. We show that network virtualization can be enabled in a way that is transparent to the end user while helping orchestrating distributed applications and improving security.

2 Converged computing

The research topics related to converged computing can be classified in three main categories: HPC in the cloud, hybrid solutions and cloud in HPC.

The first approach consists in trying to allow HPC workloads to run efficiently in the cloud, in particular using Kubernetes. Indeed, Kubernetes has quickly gained adoption among scientists wanting to leverage cloud resources for deploying computational workloads. To run efficiently, parallel HPC applications do however require specialized scheduling, resource allocation and process management which is not provided out of the box by Kubernetes. The *MPI Operator*[3] is commonly used to allow launching MPI applications on Kubernetes. To further bridge the gap with the level of support offered by HPC resource managers, multiple tools allow to easily instantiate virtual clusters within Kubernetes, in which resources are managed by an HPC scheduler such as Flux or Slurm [2] [3]. Some more tightly integrate the virtual cluster to the underlying environment by delegating resource management to Kubernetes which allows the simultaneous execution of unmodified HPC and non HPC workloads [4]. Other research aims at adapting Kubernetes scheduling and placement algorithms so that it may better handle HPC workloads [5] [6]. In [7], the authors present a solution to enable the scaling of HPC workloads on Kubernetes to allow for more elasticity and flexibility.

The second approach consists in creating hybrid architectures bridging cloud and HPC environments, with varying degrees of integration. For example, it can be done by offloading HPC workloads from Kubernetes to a remote HPC cluster through custom tools [8]. Another method relies on transparently converting Kubernetes-native workload specifications to Slurm commands for execution on HPC resources [9]. In [12], the authors present a proof of concept for building a Kubernetes cluster through Slurm allocations by creating Kubernetes agents on compute nodes that link back to an existing and long-standing Kubernetes control plane. The hybrid approach generally allows users to easily define their workflows through Kubernetes while still benefiting from the performance of an HPC cluster, and with minimal modification to each environment.

Finally, the third approach consists in extending HPC platforms with some capabilities more commonly available in the cloud. Many users benefit from the specialized tooling and support provided by HPC facilities and there is value

[3] https://github.com/kubeflow/mpi-operator

in making it easier to deploy a wider range of workloads in this familiar environment. On-premise HPC clusters also remain a more cost-efficient platform than the cloud for many large consumers of compute resources, and some of the largest scale HPC workloads have yet to be fully supported by most cloud providers [10]. In some cases, using the cloud is not even an option that can be considered, most notably for regulatory reasons.

A noticeable step in this direction is the growing adoption of containers [11]. By allowing users to package an application along with its dependencies and deploy it on any environment, containers are a great tool to test new software, use specific versions of dependencies, and generally reduce reliance on the underlying software stack. Deploying containers on HPC environments comes with its own set of challenges: containers must run within the resource constraints set by the resource manager and being able to run a container should not provide a user with any additional privilege on the host. Moreover, containers must be quickly started on thousands of nodes without overloading the shared filesystem. As the most popular container engine, Docker, did not meet these requirements, many HPC oriented container engines were developed to better integrate containers in HPC environments [12].

3 Motivation

While containers are now commonly supported in HPC environments, they are mostly used for a very specific purpose which is to facilitate the deployment of an application, along with its dependencies, as a self-contained image that relies as little as possible on the host software stack. This explains why most of the HPC oriented container engines only make use of the mount and user namespaces to make filesystems within a container independent from the host filesystems. Other namespaces such as the PID or network namespaces are typically shared with the host. This means that, with respect to networking, processes within a container are no different from processes launched directly on the host. While common practice in HPC environments, having all processes share the same network is far from ideal both in terms of security and ease of use.

HPC clusters are typically shared by a large number of users, sometimes even from multiple institutions. When users log in interactively to a cluster, they are presented with a shell environment that resembles a shell on their personal workstation and they tend to use it similarly, forgetting that the server is shared with other users which they do not necessarily trust. A very common security issue comes from starting network applications listening on localhost without authentication on shared machines such as login nodes. The very popular Jupyter notebook server used to behave in this way by default, which was a critical issue considering that it allows to run arbitrary code once connected. Many users setup similarly insecure configurations, for example launching unauthenticated remote gdb servers or establishing SSH tunnels to private resources, forgetting that they are sharing a network with sometimes hundreds of untrusted users.

Moreover, HPC users are deploying more and more complex workflows tightly integrating parallel simulations with machine learning, data analytics, or in-situ visualization. Integrating these software components increasingly requires deploying network services within the HPC cluster. For example, workflow engines such as Fireworks[4] which are required to manage these complex workflows often store workflow state using network databases. Multiple components of a workflow may need to communicate with each other at runtime or a user may want to establish an interactive connection to a dashboard or visualization tool to monitor or even steer a computation. Even assuming that all network protocols are properly authenticated, reliably deploying network services or client-server applications is difficult on an HPC cluster with a shared network. Network daemons launched by a user cannot listen on pre-defined network ports as they may already be used by another user and the address at which they are reachable can change whenever they are started on new nodes by the scheduler for example.

One way to overcome these limitations is to leverage network namespaces to setup a dedicated network stack for containers, which is what Kubernetes does for each pod (group of tightly coupled containers). Each pod is assigned its own IP address which means there is no risk of conflict in case the same port is used by another pod on the same host. Users can also associate domain names to their services, which allows them to know in advance at which domain name and port a service will be reachable, no matter the state of the cluster or on which nodes containers are effectively deployed. Network policies can be defined to isolate traffic as needed, for example between multiple tenants.

4 A Simple model for HPC clusters

To take full advantage of containers and solve the issues identified above, we propose to apply a networking model loosely inspired by Kubernetes to an HPC cluster managed by Slurm. While Kubernetes and Slurm implement very different sets of features, they both share the same core capability: allocating resources on a cluster to schedule user-defined workloads.

Using Kubernetes, the main unit of scheduling is the pod which represents a set of resources on a single server in which one or more containers are executed. Containers within a pod share some of their namespaces, in particular the network namespace. Each pod thus has its own network stack, and its own IP address. Users do not normally manage individual pods. They create workload objects such as jobs or deployments which define the containers they want to run and how many times they want to run them. Kubernetes controllers then schedule the appropriate number of pods for running these workloads.

Using Slurm, the main unit of scheduling is the job, which represent a set of resources on multiple servers in which users can run one ore more steps. A step consists in the parallel execution of multiple instances of a program, within the resources allocated for a job. By making use of an HPC container engine, these programs can be run within containers.

[4] https://materialsproject.github.io/fireworks/

In this work, we propose to use the concept of pods in a Slurm cluster. Mirroring the definition used in Kubernetes and other tools such as Podman, we use the term 'pod' to describe a set of namespaces, in our case PID and network, shared between multiple containers on a node. Similarly as with Kubernetes, users do not have to manage pods directly: they can submit Slurm jobs as usual and pods are automatically created when resources are allocated. Steps are then run in containers within the pod created for the job on each node. For a given job, a single pod is created per node but a node can be shared by mutliple pods if it has been allocated by multiple concurrent jobs.

The properties of Kubernetes pods were chosen to facilitate the transition from earlier types of clusters where related processes were run within the same virtual machine or bare metal operating system. These properties also help make the introduction of pods, along with network namespaces, transparent for typical HPC workloads. As processes started one a node for a job are executed within the same pod, they can interact with each other normally using either shared memory or loopback connections, but they are now isolated from processes of other jobs. Each pod is given an IP which allows it to communicate with other pods as well as with processes on the host without NAT. This allows processes in multi-node jobs to communicate normally with each other as well as with the Slurm daemons.

Kubernetes namespaces (completely unrelated to the container namespaces discussed until now) can be used to create naming scopes for group of resources such as pods as well as to define access policies between them. In a multi-tenant cluster, it is common to assign distinct namespaces to each tenant. By default Kubernetes allows unrestricted network communications between all pods and network policies are used to define filtering rules. A commonly applied policy consists in forbidding network traffic across namespaces.

In this work, as a first approach, we consider that each user of an HPC cluster is assigned its own equivalent of a namespace and we apply a default global network policy which only allows network traffic between pods belonging to the same user. Cluster administrators can also define global or per-user ingress/egress rules for communicating with addresses outside of the pod network. Support for more flexible definition of namespaces and network policies may be studied as future work.

As in Kubernetes, DNS records are created for each pod, in a subdomain associated to each namespace, or user in our current implementation. For each job, pod<n>.<jobname>.<user>.<cluster> maps to the pod on the *nth* node of the allocation, and <jobname>.<user>.<cluster> maps to all pod IPs. The resolver of containers started in a pod is configured to search first in their job domain, then in their namespace domain before defaulting to a set of domains configurable by cluster administrators.

HPC clusters can usually be accessed through login nodes, on which users can start interactive shells. We apply the principles described above to interactive workloads as well by automatically creating for a pod for each user when they establish their first connection to a login node. For each connection, a container is

started in the pod previously created for the user and the pod is destroyed when the user no longer has any active sessions on the node. Users can also establish SSH connections to compute nodes which are allocated to them for running jobs. For each such connection, a container is created in the pod matching the most recent job on the node for the user.

5 Implementation of Pcocc Networking

The networking model described in the previous section has been implemented in pcocc (pronounced like "peacock", for private cloud on a compute cluster), a tool which allows to run virtual machines and containers in HPC clusters. While originally written as a standalone Python executable, the container engine has been recently rewritten in Rust, with one objective being to allow tighter integration with the scheduler through the use of a Slurm SPANK[13] plugin. This plugin is used to start pods by hooking into the *extern* step which is a special step that always runs as soon as nodes as allocated for a job. It then creates containers in these pods for running processes launched by each job step, either from a user specified image or from a default image. For interactive accesses, the ForceCommand directive of the OpenSSH server allows to hook SSH connections to spawn containers for each session.

Configuring network namespaces is delegated to a network daemon running on each node of the cluster. It implements a simple protocol through a local UNIX domain socket, which allows to create or teardown a network namespace for either a job or a login pod. Each namespace is provided with a single virtual Ethernet interface. In our current implementation RDMA networks are not namespaced and can be used as normal. Isolating RDMA communications is left for future work.

The main difference between the setup of pods for job or login sessions resides in how IP adresses are allocated. For jobs, a single request allows to allocate all IPs required for a job at once. The lifecycle of these IPs is tied to the lifecycle of their job which can be queried from the Slurm controller. Login sessions only allocate IPs for a single pod. Their lifecycle is tied to a lease system which the network daemon must periodically refresh to keep the allocation. The full IP allocation state is stored in etcd for high availability and for use by network daemons.

Virtual Ethernet interfaces configured for each pod are interconnected using layer 2 tunnels encapsulated by VXLAN which are configured when pods are started on a node. A unique tunnel identifier is allocated for each user which ensures that packets cannot be forwarded between pods belonging to different users. Figure 1 depicts how these tunnels are setup. Connectivity between the pod network and external nodes is provided through gateway nodes on which the network daemon setups forwarding rules based on the etcd state. It implements global and per-user filtering for connections in and out of the pod network.

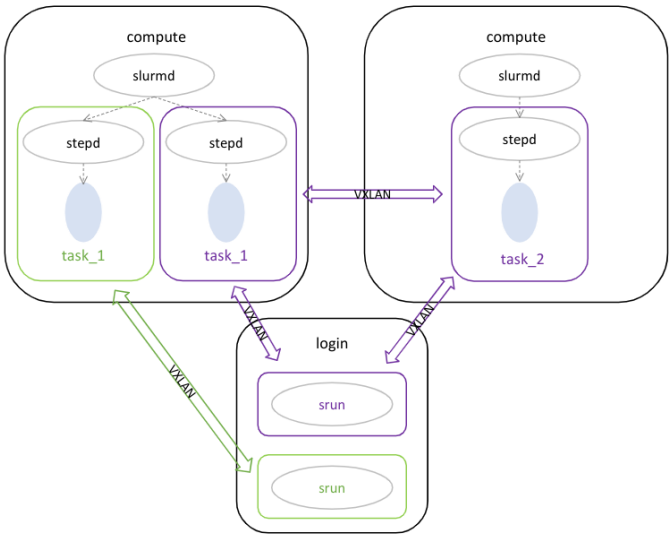

Fig. 1. Pods interconnected by VXLAN tunnels for two users executing two concurrent jobs with the Slurm srun command. Pods are depicted as colored boxes, each color representing a different user.

As can be seen on figure 1 the per-job slurmstepd daemon of Slurm is put in the same network namespace as the job. This is required as this daemon provides a PMIx server that must be accessible over localhost from the job processes.

The DNS server exposed in containers is implemented using CoreDNS. Domains are populated based on the IP allocation state stored in etcd.

This first implementation of our network model was guided by the objective of being easily deployable in an existing cluster without any disruption to non-containerized workloads. Compute nodes do not incur any additional load or complex network configuration when they are not hosting any pod and the use of an encapsulated overlay network allows some independence from the network architecture of the underlying cluster. In this work we did not leverage existing Kubernetes plugins, as custom networking logic had to be implemented in to emulate features outside the scope of the CNI and to ensure transparent operation from the point of view of Slurm. Implementing the whole network stack allowed us to more freely experiment with the design. In the future we nonetheless plan to support the CNI interface to benefit from the wide range of deployment options and features offered by Kubernetes networking solutions. In partcular, we plan to evaluate Calico which implements the container network with layer 3 routing, thus avoiding the overhead of encapsulation at the cost of a potentially more complex deployment, depending on the network layout of the cluster.

6 Performance evaluation

To evaluate the performance of our approach, we measure the overhead of running well-known HPC benchmarks relying on MPI (HPCG and HPL) inside containers deployed by pcocc. Containerization has been shown to add negligible overhead to the execution of HPC jobs [14] when sharing the host network. However network virtualization, especially when relying on encapsulation, may increase the cost of processing each network packet and reduce application performance.

First, we focus on the overhead of using containers with virtual networks at job startup. As we want to evaluate the scalability of our approach at a larger scale, tests are run in a virtual cluster of 400 virtual machines (VMs). The VMs are organized as a typical HPC cluster managed by Slurm, each VM acting as a compute node with 6 cores.

We measure the time needed to create the networking configuration for a set of pods when launching a job. When a job is launched, each node calls the local networking daemon from the Slurm plugin and waits until the network is set up. This operation, including the call to the daemon itself, is timed and printed in the logs. For each job launch, the maximum value across all nodes is recorded as the last node to finish will also be the last one to start the MPI processes so any delay would impact the whole job execution. These maximum values are averaged across 20 executions for each job size and the results are shown at the top of figure 2. The total time needed to setup the network scales well with just above 800 milliseconds needed to setup the network for a job spanning 400 nodes. Compared to the time Slurm takes to launch a job in general, which is about 300 to 500 milliseconds in this virtual environment, we believe it is an acceptable overhead as it only doubles the time needed for smaller jobs and becomes insignificant for larger jobs as compared to the expected execution time. Next, we measure the time needed to execute a job step within an existing allocation. We use the osu_init benchmark from the OSU microbenchmark suite (v7.3) and measure the total execution time of the srun osu_init command, performed in a batch script. This test emphasizes the overhead of starting containers for each execution step, and of relying on the virtual network for performing the initial wireup of the communications between the tasks during MPI_Init. MPI_Init, which has to be called at the beginning of MPI tasks, carries out a PMIx Fence operation that exchanges messages using TCP/IP and is the most likely to be impacted by network virtualization overhead in typical MPI applications. Further communications are generally performed through RDMA which is currently shared with the host.

Performance samples are collected within a batch script which times the execution of 50 sequential steps. The bottom of figure 2 shows the distribution of measured times for jobs of 50 and 400 nodes in three configurations: without containers ('native'), using containers sharing the host network ('ctr') and using containers with network virtualization ('ctrnet'). The result shows that the network virtualization does not seem to impact the execution time of a step significantly, as the results for 'ctr' and 'ctrnet' stay very similar for smaller

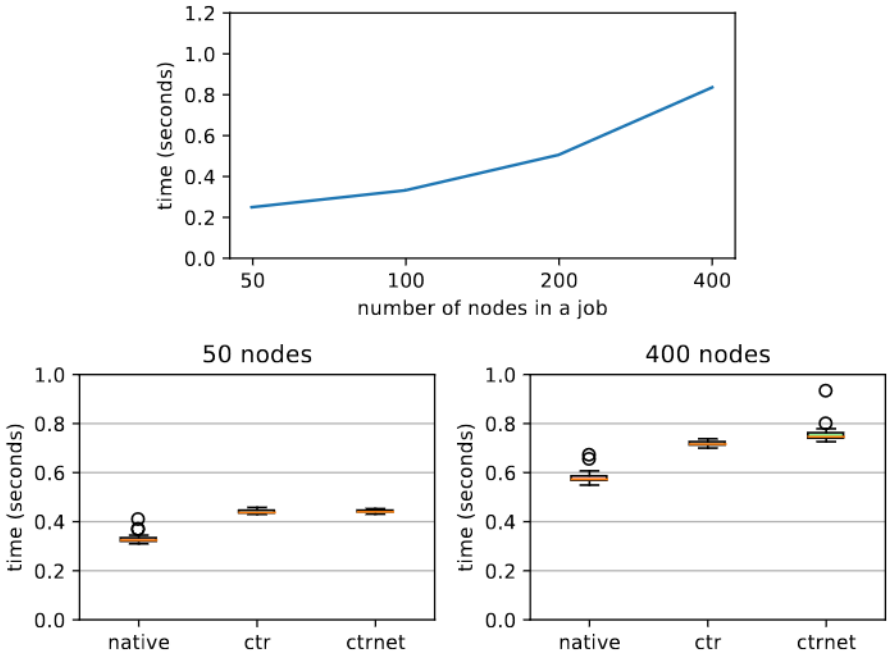

Fig. 2. (Top) Average time taken to setup the networking on a node for different job sizes. (Bottom) Execution time of a job step executing the osu_init benchmark. The nodes are virtual machines with 6 cores each.

and larger job sizes. The constant gap between the native and containerized versions correspond to the container creation step, unrelated to the networking. This comforts the idea that using the container network should not significantly lengthen the execution time of an MPI job, especially considering that only TCP/IP communications are affected.

Finally, we deployed our solution on bare-metal compute nodes to measure the impact of network virtualization on the execution of typical HPC workloads making full use of the underlying hadware. For this purpose, we use a compute partition composed of dual-socket Intel Xeon Gold 6148 CPUs for a total of 40 cores and 175GB of usable memory per node. Nodes are interconnected with an EDR Infiniband network. The software environment on the cluster consists of RedHat Entreprise Linux 8.8, Slurm 23.11.1, OpenMPI 4.1.4 and PMIx v4. We chose two popular benchmarks, HPL (v2.3)[15] and HPCG (v3.1)[16], which are commonly used to assess the performance of HPC clusters for running parallel applications. We compared the results for native, containerized and containerized with network virtualization executions on 50 nodes. To ensure comparable results, all executions of a given benchmark are performed on the same set of nodes and the same software stack is deployed on the containers as on the host.

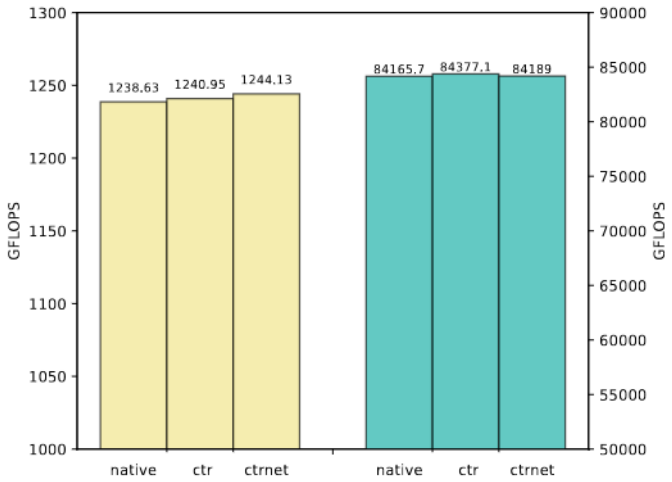

Fig. 3. Average GFLOP/s of HPCG (left) and HPL (right) on 50 nodes.

The results are shown in figure 3. The observed differences in performance are miminal, with less than a 1% overhead for HPCG and HPL with the container network. This result is consistent with what we observed on the virtual cluster, considering that only PMIx and a few communications back to the launcher use the container network.

7 Conclusion

In this paper, we show how classical HPC clusters can benefit from adopting a networking model similar to that used in cloud environments. As containers have become popular to facilitate deploying complex software stacks involved in HPC workflows, we present a solution that transparently leverages network virtualization in containers to bring more flexibility and security to HPC clusters. Isolating users in their own private network prevents unwanted accesses and facilitates the deployment of services which can be reached at known ports or domain names. We evaluate the impact of our solution on application performance by timing the network creation at the beginning of a job and the execution time of typical HPC benchmarks. We find that jobs launched with virtual networks can be started in less than a second at the scale of several hundred nodes. We also observe less than 1% overhead when running common parallel benchmarks. To further confirm these results, real applications and workflows will be deployed and evaluated. In the future, we plan to evaluate alternate implementations of this model which avoid the use of encapsulation, potentially leveraging network components used in Kubernetes. We also plan to virtualize RDMA interfaces by

making use of Infiniband partition keys so as to fully isolate all networks that users have access to.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

REFERENCES

1. Kubernetes, https://kubernetes.io/, last accessed 2024/02/22
2. Sochat, V., Culquicondor, A., Ojeo, A., Milroy, D.: The Flux Operator. early access, Oct. 2, 2023, https://doi.org/10.48550/arXiv.2309.17420
3. Greneche, N., Menouer, T., Cérin, C., Richard, O. (2022). A Methodology to Scale Containerized HPC Infrastructures in the Cloud. In: Euro-Par 2022: Parallel Processing. Euro-Par 2022. Lecture Notes in Computer Science, vol 13440. Springer, Cham. https://doi.org/10.1007/978-3-031-12597-3_13
4. Zervas, G., Chazapis, A., Sfakianakis, Y., Kozanitis, C., Bilas, A. (2022). Virtual Clusters: Isolated, Containerized HPC Environments in Kubernetes. In: ISC High Performance 2022 International Workshops. ISC High Performance 2022. Lecture Notes in Computer Science, vol 13387. Springer, Cham. https://doi.org/10.1007/978-3-031-23220-6_24
5. Milroy, D., et al. (2022). One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC. In: 2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE. https://doi.org/10.1109/CANOPIE-HPC56864.2022.00011
6. Liu, P., Guitart, J. (2022). Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters. In: IEEE 24th Int Conf on High Performance Computing & Communications (HPCC/DSS/SmartCity/DependSys). IEEE. https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00068
7. Medeiros, D., Walhgren, J., Schieffer, G., Peng, I. (2023). Kub: Enabling Elastic HPC Workloads on Containerized Environments. In: 2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE. https://doi.org/10.1109/SBAC-PAD59825.2023.00031
8. López-Huguet, S., Segrelles, J.D., Kasztelnik, M., Bubak, M., Blanquer, I. (2020). Seamlessly Managing HPC Workloads Through Kubernetes. In: ISC High Performance 2020. Lecture Notes in Computer Science, vol 12321. Springer, Cham. https://doi.org/10.1007/978-3-030-59851-8_20
9. Chazapis, A., Nikolaidis, F., Marazakis, M., Bilas, A. (2023). Running Kubernetes Workloads on HPC. In: ISC High Performance 2023. Lecture Notes in Computer Science, vol 13999. Springer, Cham. https://doi.org/10.1007/978-3-031-40843-4_14
10. Lange, J., et al. (2023). Evaluating the Cloud for Capability Class Leadership Workloads. ORNL/TM-2023/3083, Oak Ridge Leadership Computing Facility. https://doi.org/10.2172/2000306
11. Ferlanti, E., Allen, W., Lima, E., Wang, Y., Fonner, J. (2023). Perspectives and Experiences Supporting Containers for Research Computing at the Texas Advanced Computing Center. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. ACM. https://doi.org/10.1145/3624062.3624587

12. Mujkanovic, N., Durillo, J., Hammer, N., Müller, T. (2023). Survey of Adaptive Containerization Architectures for HPC. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. ACM. https://doi.org/10.1145/3624062.3624588
13. SPANK, https://slurm.schedmd.com/spank.html, last accessed 2024/03/04
14. Torrez, A., Randles, T., Priedhorsky, R. (2019). HPC container runtimes have minimal or no performance impact. In: 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE. https://doi.org/10.1109/CANOPIE-HPC49598.2019.00010
15. HPL benchmark, https://www.netlib.org/benchmark/hpl/, last accessed 2024/03/04
16. HPCG benchmark, https://www.hpcg-benchmark.org, last accessed 2024/03/04