A Methodology to Scale Containerized HPC Infrastructures in the Cloud

Nicolas Grenèche, Tarek Menouer, Christophe Cérin, Olivier Richard

A methodology to scale containerized HPC infrastructures in the Cloud

Nicolas Grenèche[1], Tarek Menouer[2], Christophe Cérin[1,3] and Olivier Richard[3]

[1] University of Paris 13, LIPN - UMR CNRS 7030, 93430 Villetaneuse, France
{nicolas.greneche,christophe.cerin}@univ-paris13.fr
[2] Umanis Research & Innovation, 92300 Levallois-Perret, France
tmenouer@umanis.com
[3] University of Grenoble Alpes, 38400 Saint-Martin-d'Hères, France
olivier.richard@imag.fr

Abstract. This paper introduces a generic method to scale HPC clusters on top of the Kubernetes cloud orchestrator. Users define their targeted infrastructure with the usual Kubernetes syntax for recipes, and our approach automatically translates the description to a full-fledged containerized HPC cluster. Moreover, resource extensions or shrinks are handled, allowing a dynamic resize of the containerized HPC cluster without disturbing its running. The Kubernetes orchestrator acts as a provisioner. We applied the generic method to three orthogonal architectural designs Open Source HPC schedulers: SLURM, OAR, and OpenPBS. Through a series of experiments, the paper demonstrates the potential of our approach regarding the scalability issues of HPC clusters and the simultaneous deployment of several job schedulers in the same physical infrastructure. It should be noticed that our plan does not require any modification either in the containers orchestrator or in the HPC schedulers. Our proposal is a step forward to reconciling the two ecosystems of HPC and cloud. It also calls for new research directions and concrete implementations for the dynamic consolidation of servers or sober placement policies at the orchestrator level. The works contribute a new approach to running HPC clusters in a cloud environment and test the technique on robustness by adding and removing nodes on the fly.

Keywords: Resource management in HPC Clusters and Clouds · Containers · Scalability · Orchestration · Aggregation and federation of HPC Clusters in the Cloud

1 Introduction

Traditionally, HPC clusters have been all about numerical simulation. Scientists and engineers would model complex systems in software on large-scale parallel clusters to predict real-world outcomes. Financial risk management, computational chemistry, omics (genomics, proteomics, metabolomics, metagenomics, and transcriptomics), seismic modeling, and simulating car crashes in software

are good examples of numerical simulations. An HPC cluster gathers hardware nodes managed by a single software called the batch scheduler. This scheduler runs scientific workloads on the hardware according to scientists' resource definition constraints. This point results in a very specialized infrastructure designed for massively parallelized applications.

Over the past decade, however, what we consider to be an HPC cluster has broadened considerably. Today, clusters are supposed to involve collecting or filtering streaming data, using distributed analytics to discover patterns in data, or training machine learning models. Usages include, nowadays, interactive workloads and even, for the data science community, "long-running" distributed services such as TensorFlow, Spark, or Jupyter notebooks. As HPC applications have become more diverse, scheduling and managing workloads have evolved. The diversity in applications pushed people to wonder if Cloud systems would be a better computer systems and even if the Cloud would encompass all the categories of scientific issues in a unified way. Our paper is a step in this last direction. It addresses the following challenge: is it possible to execute various HPC job schedulers on the same infrastructure, controlled by a Cloud orchestrator?

The Cloud orchestrator may play a similar role to the HPC batch scheduler. However, the aim is slightly different. They both place active processes on hardware resources, but these processes have a different natures. An HPC cluster is designed to run non-interactive scientific workloads with a beginning and an end. A Cloud orchestrator lets users define a targeted containerized infrastructure and endeavors to satisfy their needs, including restarting failed components. In fact, in system administration, orchestration is the management of computer systems and software, as with the batch scheduler, and the automated configuration and coordination of the computer system.

In a nutshell, we containerized several batch schedulers (OAR [4], SLURM, and OpenPBS in our experiments). These schedulers are hosted on Cloud infrastructure (Kubernetes in our experiments). We attempt to solve the problem of scaling, i.e., dynamically add or remove containerized HPC nodes (we will reference them as workers), without altering neither the Cloud orchestrator nor the HPC scheduler. This work results in a generic method to coordinate Cloud orchestrator and HPC scheduler.

We face many scientific challenges in integrating the new features described above, making the task challenging. First, the targeted Cloud orchestrator for our experiments, Kubernetes, has limited support for the HPC types of workloads[4]. Second, we must check that HPC workloads can run in containers and become Kubernetes friendly. Third, end-users (i.e., people that submit a job to the HPC scheduler) must not be aware that their computations run on a cloudified HPC cluster.

The organization of the paper is as follows. Section 2 introduces some related works on HPC cloudification. Section 3 introduce mainlines of our methodology to position our contribution among these works. Section 4 introduces exhaustive

[4] https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/

experiences that allow the validation of our proposed approach. At last, we introduce future works in section 5.

2 Related works

In this paper, we propose advanced integration into the Cloud of popular batch schedulers and discuss the suitability of the methodology for the resonance between HPC and Cloud systems. First, we added a degree of difficulty with the ability, for our proposal, to deploy and remove on the fly multiple batch schedulers. More importantly, we developed a layer between the batch scheduler and the Cloud orchestrator that dynamically adds or removes computational nodes, thanks to dedicated mechanisms at the Cloud orchestrator level. Second, we also imposed another constraint for the integration: to modify the orchestrator and batch scheduler sides as little as possible. We mean to count first on the existing mechanisms and not be intrusive in the current architectures. Notice that our work is not related to scheduling jobs or pods but to a generic interposition mechanism to glue HPC and Cloud middlewares.

The IBM Spectrum LSF Suites portfolio [6] redefines cluster virtualization and workload management by integrating mission-critical HPC environments. IBM Spectrum LSF Suites supports organizations using container technologies, including Docker, Shifter, and Singularity. This feature streamlines an application's building, testing, and shipping, enabling an application stack to be deployed on-premises consistently and in the Cloud. IBM Spectrum LSF is not devoted to the containerization of HPC job schedulers.

Kubernetes, commonly stylized as K8s [7] is an open-source container orchestration system for automating software deployment, scaling, and management. Kubernetes aimed to solve an entirely different problem than the traditional problems solved by HPC clusters – delivering scalable, always-on, reliable web services in Google's Cloud. Kubernetes applications are assumed to be containerized and adhere to a cloud-native design approach. Pods which are groups of one or more CRI-O[5] or OCI[6] compliant containers, are the primary constituents of applications that are deployed on a cluster to provide specific functionality for an application. Kubernetes provides features supporting continuous integration/delivery (CI/CD) pipelines and modern DevOps techniques. Health checks give mechanisms to send readiness and liveness probes to ensure continued service availability. Another differentiating feature is that Kubernetes is more than just a resource manager; it is a complete management and runtime environment. Kubernetes includes services that applications rely on, including DNS management, virtual networking, persistent volumes, secret keys management, etc.

In [9], the authors address the problem of running HPC workloads efficiently on Kubernetes clusters. They compare the Kubernetes' default scheduler with KubeFlux, a Kubernetes plugin scheduler built on the Flux graph-based scheduler, on a 34- node Red Hat OpenShift cluster on IBM Cloud. They also detail

[5] https://cri-o.io/
[6] https://opencontainers.org/

how scheduling can affect the performance of GROMACS, a well-known HPC application, and they demonstrate that KubeFlux can improve its performance through better pod scheduling. In contrast with our work, authors work at the level of one application (GROMACS), whereas we are working on containerizing job-schedulers.

In [2], authors studied the potential use of Kubernetes on HPC infrastructure for use by the scientific community. They directly compared both its features and performance against Docker Swarm and bare-metal execution of HPC applications. They detailed some configurations required for Kubernetes to operate with containerized MPI applications, explicitly accounting for operations such as (1) underlying device access, (2) inter-container communication across different hosts, and (3) configuration limitations. They discovered some rules that showed that Kubernetes presents overheads for several HPC applications over TCP/IP protocol.

In [12] authors argued that HPC container runtimes (Charliecloud, Shifter, Singularity) have minimal or no performance impact. To prove this claim, they ran industry-standard benchmarks (SysBench, STREAM, HPCG). They found no meaningful performance differences between the used environments, except modest variation in memory usage. They invite the HPC community to containerize their applications without concern about performance degradation.

In [16], authors describe a plugin named Torque-Operator. The proposed plugin serves as a bridge between the HPC workload manager Torque and the container orchestrator Kubernetes. The authors also propose a testbed architecture composed of an HPC cluster and a big data cluster. The Torque-Operator enables the scheduling of containerized jobs from the big data cluster to the HPC cluster.

In [13], the authors show the usefulness of containers in the context of HPC applications. They introduce the experience of PRACE (Partnership for Advanced Computer in Europe) in supporting Singularity containers on HPC clusters and provide notes about possible approaches for deploying MPI applications in using different use cases. Performance comparisons between bare metal and container executions are also provided, showing a negligible overhead in the container execution in an HPC context.

In [15] authors' main concern is to define a model for parallel MPI application DevOps and deployment using containers to enhance development effort and provide container portability from laptop to clouds or supercomputers. First, they extended the use of Singularity containers to a Cray XC-series supercomputer and, second, they conducted experiments with Docker on Amazon's Elastic Compute Cloud (EC2). Finally, they showed that Singularity containers operated at native performance when dynamically linking Cray's MPI libraries on a Cray supercomputer testbed. They also concluded that Amazon EC2 environment may be helpful for initial DevOps and testing while scaling HPC applications better suited for supercomputing resources like a Cray.

In [3] authors discuss several challenges in utilizing containers for HPC applications and the current approaches used in many HPC container runtimes.

These approaches have been proven to enable the high-performance execution of containers at scale with the appropriate runtimes.

In [14], authors introduce a technique called personal cluster, which reserves a partition of batch resources on the user's demand in a best-effort manner. One individual cluster provides a private cluster dedicated to the user during a user-specified period by installing a user-level resource manager on the resource partition. According to the results obtained in this study, the proposed technique enables cost-effective resource utilization and efficient task management. It provides the user a uniform interface to heterogeneous resources regardless of local resource management software.

In [1], the authors highlight issues that arise when deploying network address translation through containers. In this paper, the authors concentrate on Docker as the container technology of choice and present a thorough analysis of their networking model, focusing on the default bridge network driver used to implement network address translation functionality.

In[10], the authors propose to test container portability on three different state-of-the-art HPC architectures (Intel Skylake, IBM Power9, and Arm-v8) and compare three critical container implementations. From the outcomes of all this, the authors hope to provide system administrators, facility managers, HPC experts, and field scientists with valuable research for guidelines and use-case examples.

3 Methodology

This section describes our methodology from a macro point of view. In the next section, we will go further in implementation details that refer to the micro point of view. The current branch outlines the method not specifically related to the three evaluated HPC schedulers. Our explanation is divided into two parts. First, we enumerate all information users must feed to categorize their Pods. Then, we describe all underlying services that we must develop to configure or reconfigure the containerized HPC infrastructure to match the resources requested by users from Kubernetes. The term "user" relates to the person who defines and instantiates the containerized HPC cluster. The term "developer" is used to determine the person who develops services used for coupling the Cloud orchestrator and the job scheduler.

3.1 Required information at a user level

This section is all about users. They describe the Pods composing the targeted containerized HPC clusters, and these Pods can have two roles depending on the hosted service. There are two primary services: schedulers and workers. Schedulers decide where to place the jobs on the infrastructure regarding their resource constraints. The worker is a service on HPC nodes that executes the job. In our method, users must supply essential information in the Pod definition. The listing 1.1 is a shortened example of a set of definitions for Workers. Users write

these listings. Let's go on with the comment on this listing to understand the main requirements.

On line 2, we can see that Workers Pods are defined as a `StatefulSet`. A `StatefulSet` is a set of identical Pods that manages stateful applications and guarantees the ordering and uniqueness of these Pods. A `StatefulSet` contains Pods based on identical container specifications. `Statefulset` also maintains a sticky identity for each Pod. The keyword `replicas` (line 9) gives the number of instanced Pods. On line 7, the label `role` informs on the type of Pod (Scheduler or Worker). Here, we have a `worker` Pod. In HPC clusters, homogeneous nodes are frequently gathered in partitions or queues (the denomination may differ from one HPC scheduler to another one). In line 13, we label this set of Pods with `partition` set to `COMPUTE` (a partition is a set of nodes). Line 17 to 21 gives the resource constraint required by the Worker Pod to the Kubernetes orchestrator. Here, we request 2 CPUs. In a nutshell, this example instantiates two Pods with two CPUs each in the COMPUTE partition.

```
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: hpc-node
5    namespace: hpc-nico
6    labels:
7       role: worker
8  spec:
9    replicas: 2
10   template:
11     metadata:
12       labels:
13         partition: COMPUTE
14       containers:
15       - name: <my_worker_name>
16         image: <my_hpc_sceduler_image>
17         resources:
18           limits:
19             cpu: "2"
20           requests:
21             cpu: "2"
```

Listing 1.1. Example of a user-defined Worker

3.2 Configuration services

This section explains services supplied by developers. At a glance, there are two sets of services: initialization and resource polling. These two sets run sequentially, one after the other. When the initialization phase is over, the resource polling starts and lasts until the whole containerized HPC cluster revocation.

The initialization service is an `initContainer` that runs before any Worker or Scheduler Pod. An `initContainer` is a container that runs before any regular container of the Pod. Standard containers start when the `initContainer`

ends successfully, i.e., the containerized process exits with return code zero. In our method, the `initContainer` aims at bootstrapping configuration for both Worker and Scheduler Pods. From the scheduler's point of view, this bootstrapping can be mapped to a configuration of a scheduling algorithm, various spool directories, PID file location, etc. From the worker's point of view, the `initContainer` locates the Pod hosting the scheduler.

The resource polling aims to watch the `StatefulSets` set by users and translate them to the containerized HPC cluster, specifically the Scheduler Pod. The resources polling service is a program that connects to the Kubernetes API to get worker Pods' properties. In our previous example described in listing 1.1, the resource polling service will add two nodes with two CPUs each in the `COMPUTE` partition of the scheduler. This program may also restart the Scheduler Pod if it is needed. The resources polling program is embedded in a sidecar container of the Scheduler Pod. A sidecar container is a regular container that interacts with the Pod's main container(s). Most of the time, the interaction is a configuration update, which is the case here. The sidecar container updates the scheduler configuration. For instance, if the user patches his containerized HPC cluster `StatefulSets` to add a worker, the sidecar container automatically updates the scheduler configuration with this newcomer. This hint enables dynamic scaling of the containerized HPC cluster.

These two services, namely initialization and resource polling, are located in containers and deployed aside from the containerized HPC cluster. Consequently, neither HPC schedulers nor Kubernetes the orchestrator need to be modified. The only requirement for our method is adding RBAC policies to enable read access to the Pods' attributes from the Kubernetes API. The containerized HPC cluster can be instanced in a dedicated namespace to mitigate information leaks that may result from such a security policy. Figure 1 sums up all these interactions.

4 Experimentations

4.1 Outline

This section applies our methodology to the three major open-source HPC job schedulers: SLURM, OAR, and OpenPBS. We experience several scenarios to check the consequences of scaling (up or down) workers' containers. We use Kubernetes / CRIO v1.22, SLURM v21.08.5, OAR v2.9, OpenPBS v20.0.1 and OpenMPI v4.1.2. The section aims at highlighting the most relevant points of the approach, making the three implementations similar.

4.2 Micro description of the methodology

In this section, we go further in detail on our method implementation. We first discuss the specificities of each HPC job scheduler that impact our methodology. Then, we supply the scaling results of the three containerized HPC clusters.

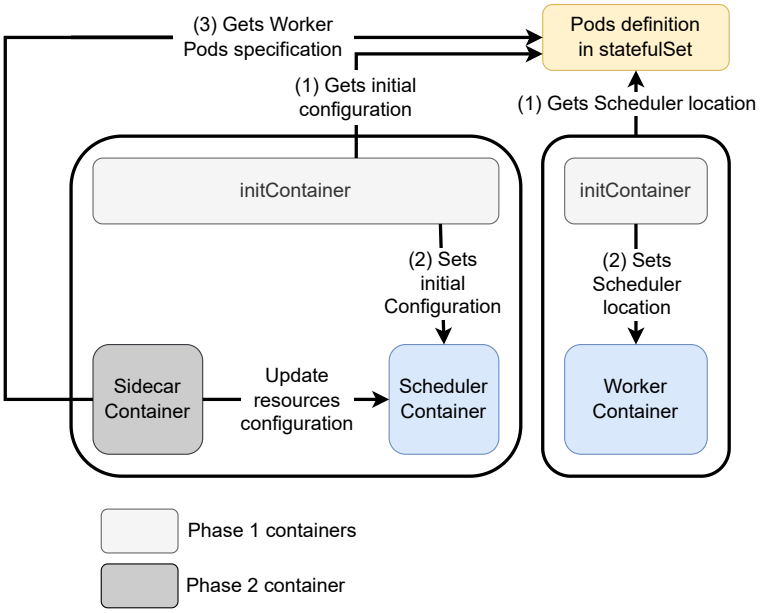

Fig. 1. Methodology: the macro level

SLURM job scheduler is built upon two services: Slurmctld and Slurmd. Slurmctld is the scheduler, and Slurmd is the worker. All HPC nodes are described in a plain text file owned by the scheduler. We configure SLURM in the configless mode: the workers connect to the scheduler to retrieve the configuration. This configuration requires the Munge service to authenticate communications between workers and the scheduler. As a result, scheduler Pod has four containers: an `initContainer`, Slurmctld, Munge, and a sidecar container that generates or updates the configuration file. The worker Pod has three Pods: an `initContainer`, Slurmd, and Munge.

Our contributions are based on introducing `initContainer` for Slurmd and Slurmctld and the Slurmctld's sidecar container. Slurmctld's `initContainer` generates a minimal configuration that enables Slurmctld to start. Slurmd's `initContainer` locates the Slurmctld service to retrieve configuration.

We have an `initContainer` for both Slurmctld and Slurmd Pods. Slurmctld's sidecar container is responsible for configuration updates when nodes are added or suppressed from containerized HPC cluster. SLURM does not support a comprehensive dynamic creation/suppression of his nodes in his current state. However, a relatively safe method is to restart Slurmctld. Then, in configless mode, all attached Slurmd daemons will reread their configuration. This method has limitations, and we will discuss them below.

Consequently, when a modification is detected in the containerized HPC cluster's topology, the sidecar container modifies the configuration file and sends a

SIGTERM signal to the Slurmctld process. We use [7] to supervise the Slurmctld process. Thus, when Slurmctld exits due to the SIGTERM reception, Daemontools' manage process restarts it gracefully without crashing the container.

OpenPBS job scheduler hosts several services. The process `pbs_sched` is the scheduler itself, `pbs_comm` handles the High Availability, and `pbs_server.bin` communicates with worker nodes to execute users' jobs. This process also interacts with a Postgres database to store resource descriptions (such as workers' specifications) and job information. We have `pbs_mom` on the worker node, which receives jobs from the PBS server to execute them on the node. The scheduler Pod has three containers: an `initContainer` that creates the configuration file for the PBS server, a container that hosts all the processes composing the PBS server, and the sidecar container that registers or unregister worker from the PBS server's database.

The containerization of OpenPBS follows the same scheme as SLURM. The `initContainer` is likely to be the SLURM's. It creates the configuration file for PBS server Pod and worker Pod. The sidecar container triggers the commands to add or delete resources in the PBS server database at each containerized HPC cluster's topology modification. OpenPBS and SLURM are very close regarding our methodology because they work on the same pattern of server/agent, and these two components are more or less coupled. We now consider a third HPC scheduler called OAR that relies on SSH for interactions between schedulers and workers.

OAR job scheduler is composed of several processes. A central one executes an automaton that reacts to all events from jobs' and nodes' states and initiates appropriate action by launching corresponding processes like scheduling round, job launching, and nodes' checking. All states related to jobs, nodes, and scheduling decisions are stored in a Postgres database. OAR is well suited for containerization because workers and schedulers are loosely coupled, making it easier to deal with synchronization. An `initContainer` in the scheduler Pod initiates a configuration for the Almighty service that drives OAR cluster resources. An `initContainer` is deployed aside from worker Pods to get the scheduler Pod location. Then, a sidecar container is executed aside from the scheduler server container inside the scheduler Pod to add or remove workers according to resources defined on the `StatefulSets`.

4.3 Experimental results

We investigate in this section some challenges of doing HPC in the Cloud. The main criterion for addressing them is the robustness of the approach because the behavior of the Cloud system and the applications running under the supervision of the HPC job schedulers is correct when dynamically adding or removing nodes attached to the HPC schedulers. We do not provide a performance metric

[7] https://cr.yp.to/daemontools.html

such as the overhead of the containerization but a measurable quality metric. The proposed approach can scale the containerized clusters dynamically without interfering with already running or scheduled user jobs.

Thus, we explored several scenarios to evaluate how each HPC scheduler behaves when resources (nodes) are added or removed. We qualify the impact on pending and running jobs. For each scenario, we submitted MPI and non-MPI jobs. The MPI job is a Pi computation with a Monte Carlo method. The non-MPI job is a multi-threaded infinite computation. The nature of jobs does not matter, meaning that jobs with MPI communication and without communication are both running correctly. We want to keep nodes busy and generate MPI communications while adding or removing workers' containers on the fly. In Table 1, we have four scenarios that are declined for each of the three evaluated job schedulers. There are two states of jobs regarding the queue of requests in an HPC scheduler: pending (the job is waiting for resources) and running (the job is running somewhere on the HPC cluster nodes). We consider the impact of growth and shrinking workers' containers for each state. In Table 1, a None value means that we do not encounter any problem, also suggesting that the execution was correct.

All the scenarios that we now detail realize a functional validation of our implementation according to our methodology for containerization. This artifact is concerned with Section 4 (Experimentations) of our Paper "A methodology to scale containerized HPC infrastructures in the Cloud". The artifact consists of a set of virtual machines from where you can deploy a comprehensive Kubernetes cluster from an Ansible receipt. Then, on this Kubernetes cluster, you can deploy three major HPC schedulers (OpenPBS, OAR, and SLURM) as a set of pods. Sample codes are supplied for each HPC scheduler to check the impact of dynamic growth or shrink of the containerized HPC scheduler on pending and running jobs. This artifact is provided as a single .pdf file containing all URLs required to set up, automatically, the experimental material that runs on the virtual machines. URLs points to an OVA file containing VMs and GitHub repositories that host the Kubernetes Ansible receipt and Dockerfiles, and Kubernetes manifests for each HPC scheduler.

4.4 Impact on pending jobs

(1) Workers addition The first scenario characterizes the state of pending jobs while worker nodes are added. We launch jobs (MPI and non-MPI) on the containerized HPC cluster to consume resources. Then, when it becomes fully occupied, we submit a non-MPI job (i.e., that does not require communications between workers). This job gets the pending state, waiting for resources. We expanded the containerized HPC cluster with additional workers. The pending non-MPI job is scheduled and ends with no errors on each evaluated HPC scheduler. To complete the first scenario, we submit again a bunch of mixed MPI and non-MPI jobs to consume all resources; then, we submit an MPI job. This job is pending.

Furthermore, we expanded the containerized HPC cluster with additional workers. The MPI job is scheduled and fails with SLURM. The reason is that the MPI job is run with srun. The srun command instantiates the MPI communication infrastructure. The first MPI job scheduled on newcomer workers fails. Then, the second will work. When new nodes are added on a SLURM cluster, a reboot of slurmctld and each slurmd service is required. Dynamic nodes addition will be fully supported in the 23.02 version of SLURM [8].

(2) Workers removal The second scenario characterizes the state of pending jobs while worker nodes are removed. We target free workers (i.e., that does not run any job). We run several jobs to keep the containerized HPC cluster busy. The idea is to have some free workers but not enough to satisfy the requirements of pending jobs. We remove these free workers from the containerized HPC cluster, and the pending jobs are not impacted.

4.5 Impact on running jobs

(3) Workers addition We launch both MPI and non-MPI jobs. While they are running, we add workers. Jobs keep running and end without any errors for each containerized job scheduler. Workers' addition has no impact on running jobs.

(4) Workers removal We launch both MPI and non-MPI jobs, but we keep some workers free. While jobs are running, we remove free workers. Jobs run and end without any error for each containerized job scheduler, and workers' removal has no impact on running jobs.

Scenarios	SLURM	OpenPBS	OAR
(1) Impact on pending jobs when resources are added	Fail	None	None
(2) Impact on pending jobs when resources are removed	None	None	None
(3) Impact on running jobs when resources are added	None	None	None
(4) Impact on running jobs when resources are removed	None	None	None

Table 1. Results of experimentation

[8] https://slurm.schedmd.com/SLUG21/Roadmap.pdf

4.6 Short-term upcoming perspectives

In our experimentation, we containerized three major HPC schedulers. We evaluated the impact of containerized workers' growth or shrink for each of them. As all scenarios went well (except for the lack of an upcoming feature in SLURM), we demonstrated the potential of building a scalable, fully containerized HPC cluster in Cloud infrastructure. Consequently, a middle-term perspective for our work is to add a controller in Kubernetes that gets the state of containerized HPC cluster's queue. The sidecar containers will act as proxies between this Controller and the HPC scheduler. In our current implementation, the sidecar container adds or manually removes resources without considering the queue's state. In [11], authors introduce fine-grain applicative metrics to autoscale pods in a Kubernetes cluster.

Similarly, a possible enhancement is making our Controller use the queue state as an applicative metric to extend or shrink the containerized HPC cluster automatically. In Figure 2 we exhibit our targeted architecture at mid-term and according to the previous discussion. The primary enhancement regarding Figure 1 is the third and fourth steps: *Gets queue state* and *Informs Controller*. These steps will supply metrics to the Controller, allowing him to decide if he must add or remove worker pods.

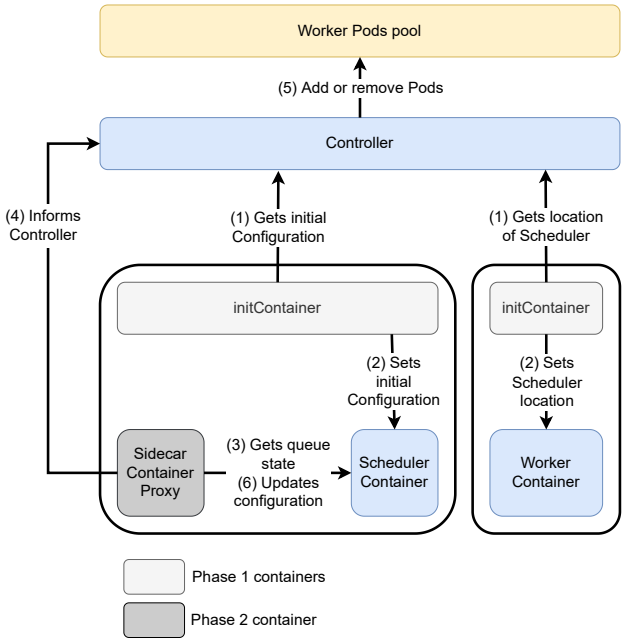

Fig. 2. Evolution of the architecture

5 Conclusion and long-term perspectives

This paper experimented with a method to build scalable containerized HPC clusters in the Cloud. We containerized three central HPC schedulers: SLURM, OpenPBS, and OAR. They all can be jailed in containers, and our experimentation demonstrates that scaling jobs do not impact running or pending jobs (except for SLURM, but this point will be handled in the upcoming release). The next step is to develop a Kubernetes controller to handle the dynamic scaling of the containerized HPC cluster. This specific contribution is part of broader work on mixing HPC and Cloud computing, and studying converged infrastructure. As an example, in [8], we developed a scheduling strategy that gathers containers belonging to the same namespace on the same node. In doing this, we concentrate our effort on scheduling issues for the server consolidation problem. Consolidation is also a tremendous problem in HPC.

Converged computing is a paradigm that aims to offer HPC performance, efficiency, and sophisticated scheduling, with cloud benefits. While orchestration frameworks like Kubernetes offer several advantages such as resiliency, elasticity, portability, and manageability, they are not performance-oriented to the same degree as HPC. Our vision of converged computing is first to put into the Cloud the HPC ecosystems and not the applications supervised by the cloud orchestrator. As pointed out above, through the example of scheduling containers in the same namespace, we separate the concerns related to containers management and scheduling and those related to the ecosystems containerization. In short, the granularity of the containerization is not the same; hence different approaches and different issues.

The implementation of a controller will also serve, in the future, to reinforce the strength and weaknesses of our approach. Moreover, it would be interesting to investigate the monetary costs of running multiple additional containers (e.g., the HPC scheduler or the sidecar container) alongside the compute node containers required for executing user applications. At last, when the controller is implemented, we will be ready to study if they are specific limits to the scalability of the proposed approach concerning scheduling options as opposed to having an HPC scheduler that handles a physical cluster. Some preliminary results show that we do not have scalability issues, but they must be comforted.

Acknowledgements and Data Availability Statement. The testbed used during the current study is available in the Figshare repository: `https://doi.org/10.6084/m9.figshare.19952813` [5].

References

[1] A. Amirante and S. P. Romano. “Container NATs and Session-Oriented Standards : Friends or Foe ?” In: *IEEE Internet Computing* 23.6 (2019), pp. 28–37.

[2] A. M. Beltre et al. "Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms". In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 2019, pp. 11–20.

[3] R. S. Canon and A. Younge. "A Case for Portability and Reproducibility of HPC Containers". In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2019, pp. 49–54. DOI: 10.1109/CANOPIE-HPC49598.2019.00012. URL: https://doi.ieeecomputersociety.org/10.1109/CANOPIE-HPC49598.2019.00012.

[4] Nicolas Capit et al. "A batch scheduler with high level components". In: *Cluster computing and Grid 2005 (CCGrid05)*. Cardiff, United Kingdom: IEEE, 2005. URL: https://hal.archives-ouvertes.fr/hal-00005106.

[5] Nicolas Greneche et al. *Artifact and instructions to generate experimental results for Euro-Par 2022 conference proceedings: A methodology to scale containerized HPC infrastructures in the Cloud*. June 2022. URL: https://doi.org/10.6084/m9.figshare.19952813.

[6] *IBM Spectrum LSF – see https://www.ibm.com/downloads/cas/VEO91OVO*. Spectrum LSF.

[7] *Kubernetes – see https://kubernetes.io/*. k8s.

[8] Tarek Menouer et al. "Towards an Optimized Containerization of HPC Job Schedulers Based on Namespaces". In: *IFIP International Conference on Network and Parallel Computing*. Springer. 2021, pp. 144–156.

[9] C. Misale et al. "It's a Scheduling Affair: GROMACS in the Cloud with the KubeFlux Scheduler". In: *2021 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2021, pp. 10–16. DOI: 10.1109/CANOPIEHPC54579.2021.00006. URL: https://doi.ieeecomputersociety.org/10.1109/CANOPIEHPC54579.2021.00006.

[10] O. Rudyy et al. "Containers in HPC: A Scalability and Portability Study in Production Biological Simulations". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2019, pp. 567–577. DOI: 10.1109/IPDPS.2019.00066. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPS.2019.00066.

[11] Salman Taherizadeh and Vlado Stankovski. "Dynamic Multi-level Auto-scaling Rules for Containerized Applications". In: *The Computer Journal* 62.2 (May 2018), pp. 174–197. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxy043. eprint: https://academic.oup.com/comjnl/article-pdf/62/2/174/27736749/bxy043.pdf. URL: https://doi.org/10.1093/comjnl/bxy043.

[12] A. Torrez, T. Randles, and R. Priedhorsky. "HPC Container Runtimes have Minimal or No Performance Impact". In: *2019 IEEE/ACM Interna-*

tional Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2019, pp. 37–42. DOI: 10.1109/CANOPIE-HPC49598.2019.00010. URL: https://doi.ieeecomputersociety.org/10.1109/CANOPIE-HPC49598.2019.00010.

[13] V. Sande Veiga et al. "Evaluation and Benchmarking of Singularity MPI containers on EU Research e-Infrastructure". In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2019, pp. 1–10. DOI: 10.1109/CANOPIE-HPC49598.2019.00006. URL: https://doi.ieeecomputersociety.org/10.1109/CANOPIE-HPC49598.2019.00006.

[14] Yang-Suk Kee et al. "Enabling personal clusters on demand for batch resources using commodity software". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–7.

[15] A. J. Younge et al. "A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2017, pp. 74–81. DOI: 10.1109/CloudCom.2017.40. URL: https://doi.ieeecomputersociety.org/10.1109/CloudCom.2017.40.

[16] Naweiluo Zhou et al. "Container orchestration on HPC systems". In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE. 2020, pp. 34–36.