Kub: Enabling Elastic HPC Workloads on Containerized Environments

Daniel Medeiros, Jacob Wahlgren, Gabin Schieffer, Ivy Peng
Department of Computer Science
KTH Royal Institute of Technology, Sweden
{dadm, jacobwah, gabins, bopeng}@kth.se

***Abstract*—The conventional model of resource allocation in HPC systems is static. Thus, a job cannot leverage newly available resources in the system or release underutilized resources during the execution. In this paper, we present *Kub*, a methodology that enables elastic execution of HPC workloads on Kubernetes so that the resources allocated to a job can be dynamically scaled during the execution. One main optimization of our method is to maximize the reuse of the originally allocated resources so that the disruption to the running job can be minimized. The scaling procedure is coordinated among nodes through remote procedure calls on Kubernetes for deploying workloads in the cloud. We evaluate our approach using one synthetic benchmark and two production-level MPI-based HPC applications – GROMACS and CM1. Our results demonstrate that the benefits of adapting the allocated resources depend on the workload characteristics. In the tested cases, a properly chosen scaling point for increasing resources during execution achieved up to $2\times$ speedup. Also, the overhead of checkpointing and data reshuffling significantly influences the selection of optimal scaling points and requires application-specific knowledge.**

***Index Terms*—HPC, Cloud, scaling, Kubernetes, Elasticity, Malleability**

I. Introduction

In recent years, two main trends have contributed to the rising importance of the convergence between High-performance computing (HPC) and cloud. The first trend is the increased compute resources on recent HPC systems, where a single node may be equipped with multiple high-end CPUs, GPUs, and 200-500 GB RAM. For instance, the Frontier supercomputer has four GPUs and 1TB memory per node. Therefore, the conventional coarse-grained static resource allocation strategy on HPC systems faces challenges in resource utilization, and efforts in exploring fine-grained dynamic resource allocation strategies that have matured on the cloud are increasing. Secondly, workloads are evolving towards more complex patterns, e.g., workflows [1], [2] incorporating traditional scientific simulations, and machine learning and in-situ data analytics. Meanwhile, the availability and accessibility of compute resources in public cloud providers like Amazon, Google, and Microsoft have attracted users to explore running HPC applications on the cloud, and previous limitations of scaling up HPC applications on cloud infrastructure are being addressed in recent active research development [3]–[6].

As user-facing cloud services tend to experience spikes or cyclical demand in their time series, cloud-native workloads, such as search and web serving, have shifted their development from a monolithic to a microservice, loosely-coupled structure. This shift was made to leverage the native adaptive autoscaling capabilities that are built into many deployment systems. These scaling decisions may be grounded in models that take into account application-specific metrics, such as the incoming number of requests, response time, available system resources (i.e., CPU, memory), and historical patterns [7].

Elasticity, which refers to the degree to which a system can adapt its resource provision to workload changes [8], has become a promising direction for converged cloud and HPC computing in recent years. In HPC systems, resource allocation to jobs is coarse-grained and static, so a job with high peak usage of resources may have to wait a long time until all resources become available for its execution. From a resource management perspective, this can block spare resources in an HPC system until a large task can start. In contrast, cloud-based systems offer fine-grained dynamic resource allocation to cloud-native workloads. This is because the cloud approach for resource allocation has developed and matured over the years to support the demand for elastic execution from cloud-native workloads. Specifically, HPC users and developers are seeking opportunities in elastic execution for high computing power, high availability, and cost efficiency.

Certain types of HPC workloads, such as deep learning, can make extensive use of this elasticity. This owes to the fact that such workloads and related applications (i.e., TensorFlow and PyTorch) are already structured in a way that shares the training/predicting load in batches among the active nodes. This makes solutions such as Elastic Horovod[1], which detects new nodes in real-time, feasible to use. However, the same cannot be directly applied to tightly-coupled workloads that are built atop Message Parsing Interface (MPI) for distributed-memory programming. Traditional HPC schedulers, such as SLURM, do not support the dynamic change of nodes in their resource allocation to a job [9]. Once a job starts running,

This research is supported by the European Commission under the Horizon project OpenCUBE (GA-101092984).

[1] Elastic Horovod: https://horovod.readthedocs.io/

adding new node resources is cumbersome, if not impossible. This means that a full start/stop process is necessary for changing the number of nodes, and the previously allocated resource will be lost, and time should be spent waiting for a new allocation if that was not previously available.

This work aims to address the problem of elastic scaling of HPC workloads by reusing the already provisioned infrastructure, with a focus on the cloud-containerized environment. We achieve this through the usage of the de-facto industry standard container orchestrator Kubernetes and multiple representative HPC workloads that allow the understanding of the gap from achieving elastic execution.

In this paper, these are our major contributions:

- We identify technical challenges of running MPI-based applications with a state-of-art container orchestrator.
- We propose an approach for elastic horizontal scaling of tightly-coupled workloads, in particular ones that are using MPI in a containerized context.
- We implement the proposed approach and quantify the overhead and its benefits in the HPC applications GROMACS and CM1.
- We discuss the application requirements and the trade-offs of elastic scaling of HPC applications in containerized environments.

This paper is structured as follows. Section II discusses the current state-of-art means for executing HPC workloads on the cloud, including some cloud-first technologies that are later used in this paper. Section III shows our approach for elastic scaling workloads and the details of our implementation. Section IV dives into the specifics of the applications workloads we are using for this work, while Section V displays our setup and results. Section VI briefly discusses some related works while Section VII consists of the conclusions and our future works.

II. Background

In this section, we describe the differences between HPC and Cloud workloads and introduce the building blocks for enabling elastic execution in this work.

A. HPC and Cloud Workloads

Cloud computing applications tend to be loosely-coupled and fault tolerant [10]. User-facing applications should be able to scale up and down according to demand, and techniques such as load balancing, where the processing batch can be directed to a less-stressed node, helps with the design of such applications. Some of the representative cloud workloads are search, data streaming, web serving, and in-memory databases [11].

Meanwhile, high-performance computing workloads tend to be tightly-coupled as there is usually an interdependence between the calculations being distributed among the nodes. For instance, the CORAL-2 benchmark suite contains representative HPC workloads[2], which includes molecular dynamics, quantum Monte Carlo, fluid simulation and cosmology.

The Message Parsing Interface (MPI) is the dominant approach for communication in HPC workloads, as a means for performing distributed calculations over a large number of compute nodes. In many cases, these calculations are explicitly or implicitly blocking operations because their execution time is determined by the slowest node due to data dependencies. Although MPI has introduced some mechanisms recently to support dynamically adding or removing members to a communicator, the schedulers on HPC systems have little support to change node allocation to a job once it starts. Therefore, in practice, this means that the only way of changing the number of allocated nodes during the execution is through the process of restarting the application.

B. Containers and Orchestrators

Containers are a solution to isolate the resources of tasks executing in the same node. In Linux-based systems, the isolation of resources is implemented through the `cgroups` feature. Docker is the industry standard for containers; however, due to security concerns (i.e., the container runs as root by default), other container technologies such as Singularity [12] and Podman [13] have been more used in HPC systems. In the case of Docker, the container is defined by a Dockerfile which contains the base image (i.e., an operating system, such as Alpine Linux or Debian) to be used and a series of deterministic commands related to the application that one desires to deploy - this includes the installation of dependencies, compilation of software/libraries and setting environment variables.

These images are deployed by orchestrators, which are responsible for not only distributing the containers among the nodes, but also monitoring them and ensuring that their characteristics - such as minimum number of replicas - stay consistent with the desired number by the user. Some of the popular orchestrators are Kubernetes, OpenShift Container Platform and Docker Swarm, with all of them being tested in HPC environments [14].

A Kubernetes cluster consists of at least two nodes, one being the master and the other being a worker node. The former runs decoupled applications responsible for the communication interface between the user and the cluster (named api-server), the scheduler, an etcd object storage for storing data and metadata from the cluster and a controller manager. Inside every worker node, there is an application named kubelet, responsible for dealing with incoming requests from the api-server, such as executing pods and exposing pod metrics for scrappers.

The basic unit within a Kubernetes cluster is the pod - an abstraction of resources for executing a container. Computational resources are defined as the time used by the CPU and the Memory. A worker node may contain one or more pods at the same time, and similarly, a pod may have one or more containers executing within the same resource domain. A set of equal pods can be associated as a ReplicaSet, where the controller manager ensures that the desired number of pods will be running in case of failure.

[2] Coral-2 benchmarks: https://asc.llnl.gov/coral-2-benchmarks

Deployments are an extended ReplicaSet with more useful features.

Architecturally, the idea of a pod was developed for tasks that should run for an indefinite amount of time. For time-limited, finite tasks, the concept of jobs is used. In practice, a job is marked as completed when a certain number of pods have been executed and successfully finished. In this case, the pods do not necessarily need to execute and finish at the same time, although in some cases (e.g., MPI-based workloads) they do.

By default, each pod has its own IP inside the Kubernetes network and can freely communicate with other pods unless otherwise defined. Aside from the IP, a pod can reach others through the usage of the built-in domain name server records which is deterministic and based on variables such as the name of the pod and its namespace. However, as either the IP or the DNS record might be mutable, Kubernetes introduces the concept of service as a means to gather a set of pods providing similar service (i.e., user-facing) and, at the same time, exposing it outside the network.

Any kind of Kubernetes-defined or custom-defined resources are defined through the usage of YAML files. The schema used by such files through Kubernetes is standardized and checked for errors before execution. Upon acceptance of the YAML file, the default Kubernetes scheduler first looks for feasible nodes and then, among the returned group of nodes, checks for the most viable ones through scoring.

Finally, the security mechanism in Kubernetes is performed through a system called RBAC (role-based access control). One may define a certain role that is able to get, create, delete or edit certain types of resources and then assign such role to, for example, a pod. By default, a pod is not able to create other pods or modify cluster-level settings.

C. Autoscaling

In cloud settings, autoscaling means changing the amount of resources of an existing allocation. This may be through the change of already allocated CPU and memory (in this case, vertical scaling) or through the increment or decrement of the number of available nodes for the application (horizontal scaling).

Kubernetes provides a built-in vertical pod autoscaler (VPA) and a horizontal pod autoscaler (HPA) by default. However, as these tools were originally designed for cloud applications, there are some problems when trying to use them with HPC workloads. First, the VPA consists of three components, namely the admission controller, recommender and updater; based on the historical pattern of CPU/Memory usage of the application, the recommender outputs a value for CPU and Memory. If the recommended value is too different from the used one, the pod is evicted and restarted. This is because Kubernetes currently does not allow changes in the requested resources of a pod unless the pod is restarted. While there are some ongoing works to address this limitation and change the allocated CPU/Memory resources dynamically, this is currently not in production versions.

The built-in HPA queries the resource utilization periodically and according to user-defined policies, such as the threshold for a certain metric, it decides to scale up or down. The metrics can be either directly related to physical resource usage (i.e., CPU or Memory) or application-level metrics, such as the number of incoming requests. However, in tightly-coupled workloads, the built-in horizontal scaler has no effect at all. If one is running an MPI application, for example, the newly started rank would execute the same calculations by itself, being unable to join dynamically the already existent communicator.

D. Volcano and MPI

Volcano[3] is a batch system for Kubernetes, providing tools for certain types of workloads that run on frameworks such as TensorFlow, Spark and MPI. Kubeflow[4] is another framework, with a strong focus on machine learning, that enables one to run MPI workloads on the cloud.

The YAML file for deploying a Volcano MPI job consists in defining a `VolcanoJob` with two types of pods: one named `master` and another that has a `worker` suffix. Additionally, the YAML file also specifies two plugins that are necessary to run MPI workloads, the `svc` and the `ssh` plugins. The former is responsible to enable all the pods within a job to visit each other by domain name and *by default* establish a policy of not allowing any other pod outside service to communicate with that network of pods. Additionally, it creates a list of all the working pods that will execute. The second plugin generates a key pair locally and mounts `/root/.ssh` as a read-only directory at every Volcano-created pod. This ensures that all the pods will have `ssh` passwordless authentication between each other, a prerequisite to smoothly run MPI jobs. There is currently an "MPI plugin" for Volcano which can be used instead of the `svc` and `ssh` plugins, in practice, replacing both, but not allowing the same degree of flexibility required by this paper.

OpenRTE [15], part of the Open MPI project, is the heart of how Volcano executes its MPI jobs. As its name implies, OpenRTE is a runtime environment that provides services related to process management, communication coordination and resource allocation. When launching a process through the `mpiexec` wrapper, one may specify which hosts are necessary to connect - this is collected by Volcano, and OpenRTE interacts with a daemon at every node - named `orted` - through a defined communication protocol to coordinate the launch and information such as launch path, environment variables and command line arguments. OpenRTE and orted keep an interaction for message passing and monitoring until the end of the execution.

Figure 1 displays the structure of a Volcano system. The master pod is responsible to start the MPI jobs, while the workers execute the workload itself. In general, the

[3] Volcano: https://volcano.sh

[4] Kubeflow: https://www.kubeflow.org/

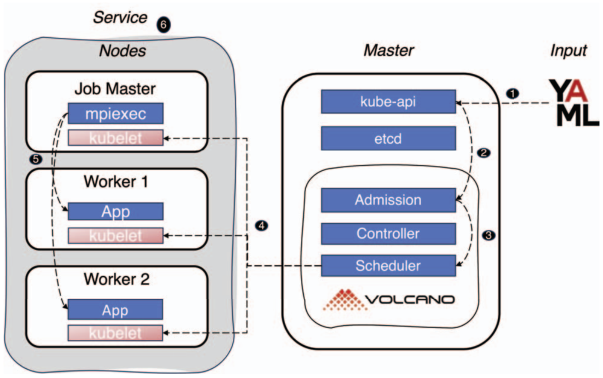


Fig. 1: The overall structure of a Volcano deployment in Kubernetes. ❶ Defined through a YAML file, the job is communicated to the api-server (or kube-api). ❷ This call is intercepted by Volcano's admission controller that checks whether the YAML has all required fields and contains no error. ❸ If everything is correct, data is sent to the scheduler which verifies whether it is possible to allocate the job. ❹ When the resources are available, the kubelets inside each node are ordered to create the pod allocation with the desired containers for the job. A pod can contain one or more containers, and nodes can also run more than one pod. ❺ The "master" pod awaits all "worker" pods to be active and open their `ssh` daemon. When they are ready, orted starts the MPI job among them. ❻ All the pods are encapsulated by a "service" type, so they can communicate using each other by domain. Volcano's scheduler and controller, responsible for monitoring the jobs, effectively replace the ones included in Kubernetes by default.

`worker-0` pod is considered the root rank of the MPI execution. While the spawning process of containers might take longer for some, the master node will keep crashing and restarting until `orted` can successfully connect with all the listed nodes. After the execution, the job is marked as Complete at the Kubernetes cluster.

E. Resource Monitoring

The kubelet that runs on every node collects information regarding CPU and Memory through cAdvisor [16], a profiling application maintained by Google that is intended for Docker containers. The Prometheus Operator[5] is a Kubernetes application that is able to scrape metrics not only from all the kubelets, but also from the master node, exposing them for further usage in JSON format. By connecting to the web API of Prometheus, one is capable of retrieving the time-series and history of the metrics, as well as checking the current status of a node in near-real time. Other applications that are combined with Prometheus are Grafana (for visualization) and the Elastic stack (Kibana and Elasticsearch). Jaeger[6] is another library for instrumenting and exposing application-level metrics in cloud applications, that can also be scraped by Prometheus.

III. Methodology

As to enable tightly-coupled workloads to be elastic, we introduce *Kub*, an extension to Kubernetes written in Python.

A. Architectural Design

Kub is composed of three major components: the Monitor, the Coordinator and the Executors. We choose to use these terms as a means to separate from Kubernetes terminology (i.e., master and workers). The Monitor is responsible for deciding when to scale as well as the creation of new pods. We understand that the decision to scale should be application-specific and left to the developers as there are trade-offs regarding the restart of a job, as it will be discussed in Section V. Due to the need for the privileges for pod creation, the Monitor may be deployed either inside the Kubernetes realm or outside. When deployed as a Kubernetes pod, it is necessary to modify the RBAC permissions from the Monitor pod so it is able to use the Kubernetes API to deploy other pods. If outside the Kubernetes realm, it is necessary to ensure that it has enough access to do so, usually granting it access to the configurations present in the host system.

In a nutshell, *Kub* works by being the process #1 when the pod is started. The rationale behind that is because, in traditional MPI applications running on Kubernetes, where Volcano is used, the launch of a `mpiexec` application as process #1 means that the job is deeply related to the status of the job, failing or completing according to it. Here, we use *Kub* for coordinating the resources for an application restart, to avoid time wasted in restarting all the infrastructure.

The checkpointing and restarting procedure is application-specific and should be written according to the application's needs. First, during checkpointing, we observe two major patterns. Some production-level HPC applications designed for long runs have already had support for checkpointing the files upon the receiving of a SIGTERM signal. Others checkpoint at each user-defined time interval. For the restarting procedure, it is necessary to handle the change of parameters in input files or a change of parameters in the command line to specify which checkpoint file should be used.

The Coordinator runs at the master pod in Volcano and effectively acts as a gRPC server. Its main purpose is for coordinating the launching of MPI applications and the eventual restarting procedure (i.e., it does not perform calculations), thus it is very lightweight. The Monitor acts as a gRPC client and its role is to define when the criteria of when the scaling process will take place, to create new pods and to tell the Coordinator how many new pods should be expected.

In a similar fashion, the executors are the worker pods as described by Volcano, or it can also be a newly-created pod by the Monitor (which we define here as a "scaling pod"), also being gRPC clients to the Coordinator. Every timestep (usually 10 seconds), they check the status of the job with the coordinator. This is done to avoid the automatic completion

[5] Prometheus Operator: https://prometheus-operator.dev/

[6] Jaeger: https://www.jaegertracing.io/

of the job when the application is paused, which can be due to checkpointing in some cases.

B. Elasticity

During the horizontal scaling of the application, when one or more pods are intended to be added, it is necessary to coordinate among all the already existing ones. The Monitor sends a message to the coordinator with the intended number of nodes to scale. As executors check the job status with the coordinators periodically, the latter changes (figuratively) its own job status to "Scaling". With such a message, the executors keep waiting for the checkpointing and the scaling process to be done instead of finishing the execution. The monitor then proceeds to create the necessary number of pods.

Figure 2 displays the entire flowchart when performing horizontal scaling. The newly-started pods cannot enter into the MPI network by default as orted cannot reach them without their IP addresses. Thus, when an additional pod is starting, it sends a message to the coordinator with its IP address and requests the current key pair shared among all the pods and, upon received, the scaling pod uses it to allow the passwordless login through orted. The coordinator keeps track of the received IP and adds it to the list of available pods to run. Finally, the job can be restarted. The process repeats when it is necessary to increase the number of pods again.

C. RPC Calls

There are many information exchanges between the master node and the monitor or the worker nodes. In this work, we use gRPC[7] – a library developed and maintained by Google which aids in the process of sending and dealing with RPC calls between applications. In practice, gRPC works through the concepts of protocol buffers: a protocol that enables serialization/deserialization of messages in many languages. By writing the message code directly into the protocol language, the message can be converted (and thus used) by languages such as Python, C++, Javascript and Go. All gRPC clients include a stub, which contains all the available remote procedures and is used to send a command to the server through a channel (usually a single HTTP 2.0 connection), and concurrent calls may be multiplexed into that channel. The server has threads waiting for any connection and will handle all necessary commands according to the procedures written in the code, returning a message thereafter. Table I displays all the calls, along the parameters, that are used in *Kub*.

D. Deployment

The deployment of *Kub* is done through the usage of a launcher script, written in Python, that is used for both the Coordinator and the Executors. Based on the standardized hostname, the launcher discerns the difference between each other and branches the code. The Monitor is also packaged as a Python script and can be run either outside or inside a pod, as described by Subsection III-A.

We focus on the horizontal scaling in this work as a first step for enabling elastic MPI-based HPC applications. The vertical solution was envisioned for single-node workloads that rely on, for example, OpenMP for thread parallelism. Our future work will investigate the vertical scaling. The horizontal scaling enabled by *Kub* is described as follows. When running an HPC workload, a pod should already default to use the maximum available resources available from a node. If it is not available, one can use the horizontal scale to increase the number of MPI ranks, with the new one using the spare resources at the moment.

E. Coordinating pods

Aside from the elastic scaling, there is an extra benefit to our approach. In Volcano, the Master pod has the #1 process running `mpiexec`, while the workers execute a `ssh` daemon. A problem happens when the Master pod executes before all workers are up. When this happens, the master pod crashes due to not being able to reach all specified pods. The master pod should then be restarted, in which it will re-execute its command in the hope that all pods will be active.

Kub avoids this problem by having the Coordinator acknowledge all the initial executors as active before starting the `mpiexec` application. As the Coordinator knows the number of initial executors, it will also expect to receive a similar amount of `JobInit` RPC calls. In the case of the Executors trying to communicate before the gRPC server is up, the fault tolerance is handled in such a way that it will retry sending the message until a response from the Coordinator is received.

IV. Applications

In this section, we characterize all the target applications for this work. In particular, there is a focus on their checkpointing process and how they were adapted to be used together with *Kub*.

A. CM1

The Cloud Model 1 (CM1) [17] is a numerical model for idealized studies of the atmosphere with a focus on deep precipitating convection. It is actively maintained by the National Center for Atmospheric Research (NCAR).

Spanning over 230,000 lines in total, CM1 is written in Fortran and supports either OpenMP, for shared memory computations, or MPI, for distributed memory. The input for the application is done through the usage of a `namelist.input` file which contains a very large number of parameters regarding the model to be simulated.

In this paper, we use the default workload for CM1 that is provided with the default `namelist.input` file. Similarly, we chose to use MPI for the calculations. For MPI, the root rank is the one responsible for distributing the load among all the available ranks and dealing with I/O

[7] gRPC: https://grpc.io/

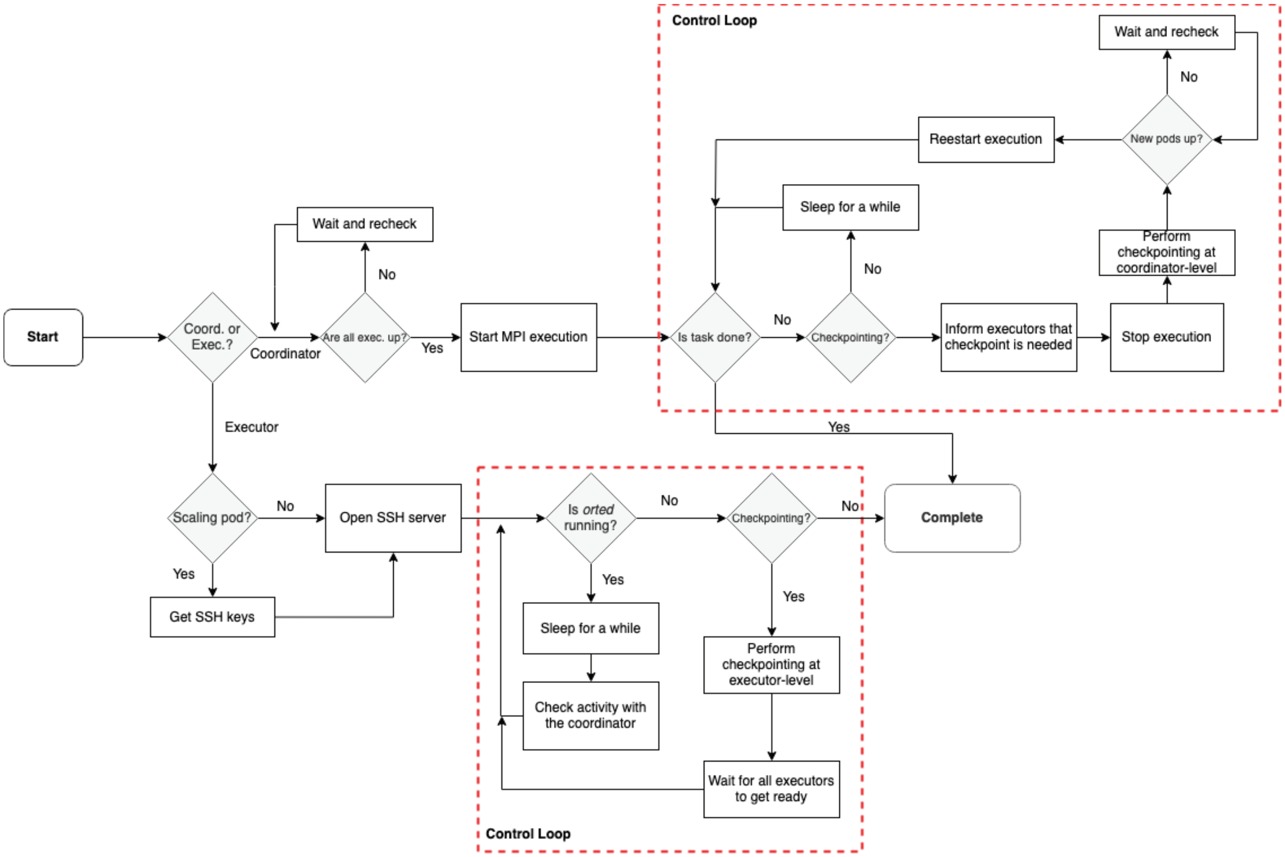


Fig. 2: The decision flowchart of *Kub* when executing. One of the major benefits of Kub is to be able to use non-provisioned infrastructure as any newly-created pod can decide to join the others by exchanging SSH keys with the Coordinator. Furthermore, the control loops ensure that the scaling can be performed multiple times during the application's execution time.

operations, which includes checkpointing and writing the final results as well. In practice, these characteristics allow CM1 to start the calculation with certain numbers of ranks and finish it with a different number.

Checkpointing. Every predefined number of timesteps, the root rank of CM1 outputs multiple checkpoint files that start with the prefix `cm1rst_` followed by the number of the checkpoint in cardinal order. The usage of this checkpointed file should be explicitly handled by the `namelist.input` file through the `irst` parameter.

Algorithm. The application-specific algorithm in *Kub* consists in checking whether the root rank is executing the code. If it is, then it iterates over the entire checkpointing directory and its related files (that starts with `cm1rst_`), looking for the most recent checkpoint based on the filename. With the id found, a small text operation for replacing the `irst` parameter on the `namelist.input` is done. The application is then ready to restart with a new number of ranks.

Modifications. There were no modifications in the vanilla code of CM1 aside from the Makefile being changed for the selection of the OpenMPI compiler wrappers.

B. GROMACS

GROMACS [18] is an open-source software suite for molecular dynamics simulation. One of its major popular use is as the backend for the distributed protein folding in the Folding@Home[8] project.

As a command line application, GROMACS is built entirely in C++ and supports a wide range of parallel and accelerating technologies, such as OpenMP and its built-in threading, MPI and GPUs (through the SYCL library). There is also support for SIMD intrinsics such as AVX-256 and AVX-512.

At the core of GROMACS is the `mdrun` engine, responsible for not only executing molecular dynamics calculations but also stochastic dynamics and energy minimization. It takes a wide range of parameters as input. It is important to mention that, due to the intrinsic randomness of the calculations, two GROMACS simulations are unlikely to yield the same results (although both of the results will be correct), even after stopping and resuming the same simulation.

[8] Folding@Home: https://foldingathome.org/

TABLE I: List of RPCs and its parameters that are used in *Kub*

Call Name	Parameter(s)	Direction	Description
Scale	Number of Nodes, Mode of Scaling	Monitor ⇒ Coordinator	This call tells the Coordinator that there are available resources and that the Job should get ready to scale.
RetrieveKeys	Name of Node	Scaling pod(s) ⇒ Coordinator	A newly-started pod can ask the Coordinator for its public and private key as to establish a ssh connection for orted.
JobInit	Name of the Node	All pods ⇒ Coordinator	This is used by the executing pods to tell the coordinator pod that the current pod is alive and ready for execution.
activeServer	(None)	All executors ⇒ Coordinator	This checks whether the master is alive and whether the working node should do any client-side checkpointing.
checkpointing	(None)	All executors ⇒ Coordinator	This is used to confirm that the checkpointing was done by the pods.
endExec	(None)	All executors ⇒ Coordinator	This is used to confirm that the execution is about to finish on the active executors.

GROMACS has been previously tested in a cloud setting [19], in particular through the usage of the AWS heterogeneous clusters (ARM, Intel, AMD CPUs and different GPUs) scattered over the world, managed with Hyperbatch and aided by the Elastic Fabric Adapter for communications between nodes located in different regions. S3 was used for storing intermediate files.

As a workload for this paper, we use one of the molecular dynamics benchmarks provided by the Max Planck Institute for Multidisciplinary Sciences[9], namely the `benchMEM` (82 000 atoms, protein in membrane surrounded by water) benchmark.

Checkpointing. The checkpointing in GROMACS is handled automatically when it receives a SIGTERM signal, writing the files as soon as it is received and gracefully stopping the application. The root rank is responsible for writing the checkpointing files and also initially reading them, distributing the data among all the available ranks.

Algorithm. The application-specific algorithm consists in sending a SIGTERM to GROMACS when it is time for checkpointing, waiting for it to write the files and killing the application. To re-execute the application from checkpointed data, an additional flag is introduced into the running command.

Modifications. No modifications in the GROMACS code were performed for this paper. The application was built according to its documentation, with the flags to build using MPI and to use its own FFTW.

C. PARINT benchmark

We design PARINT, a parallel distributed benchmark with configurable arithmetic intensity. Arithmetic intensity is the number of arithmetic operations per byte loaded from memory and measures the balance between compute and memory demands in an application [20]. PARINT consists of an outer loop over an array, with a variable number of operations per array element determined by the parameter NLOOP. With NLOOP=1, the arithmetic intensity is low and the workload is bound by the available memory bandwidth, while a high value of NLOOP gives a high arithmetic intensity, scaling with available compute power. PARINT is implemented in C and parallelized using MPI.

Checkpointing. We implement checkpointing in PARINT upon receiving a SIGUSR1 signal. Upon the arrival of this signal, PARINT will checkpoint and gracefully exit. On startup, PARINT checks whether a checkpoint file exists and loads it into memory before continuing with the main loop. The cost of checkpointing depends on the data size, determined by the `ARRAY_SIZE` parameter.

Algorithm. The algorithm consists in propagating a SIGUSR1 signal to PARINT upon receiving the call for increasing the resources. PARINT then checkpoints and the coordinator waits for the new pods to start before restarting the execution.

V. Evaluation

A. Infrastructure

In this study, we use a single-node cloud testbed that consists of an Intel i7-7820X processor, with 8 cores (16 logical cores) in total, and 32 GB of DDR4 memory at 2133 MHz. In terms of storage, the system contains an Intel Optane SSD 900p with 480 GB, a Kingston UV400 SSD, 2x Seagate Barracuda with 2 TB each and a Samsung EVO NVMe driver with 1 TB, which the operating system (Ubuntu 22.04) is running. This cloud testbed also includes an Intel I219-V single-port 1 gigabit Ethernet controller.

Furthermore, we use Kubernetes v1.23 which is deployed through k3d[10] as it emulates a single-node system. k3d includes by default the services for domain name resolution (CoreDNS), networking (traefik) and monitoring (metrics-server). Furthermore, Volcano v1.7.0 was deployed in that system.

By using a single node to run multiple pods, possible delays due to communication are mitigated and the focus shifts to the methodology itself.

All the applications were compiled using GNU Compiler Collection (GCC) v11.3 together with OpenMPI 4.1 when it was necessary as a dependency. Python 3.10 was used both on containers and system-wise for running our launcher.

[9] benchMEM: https://www.mpinat.mpg.de/grubmueller/bench

[10] k3d: https://k3d.io

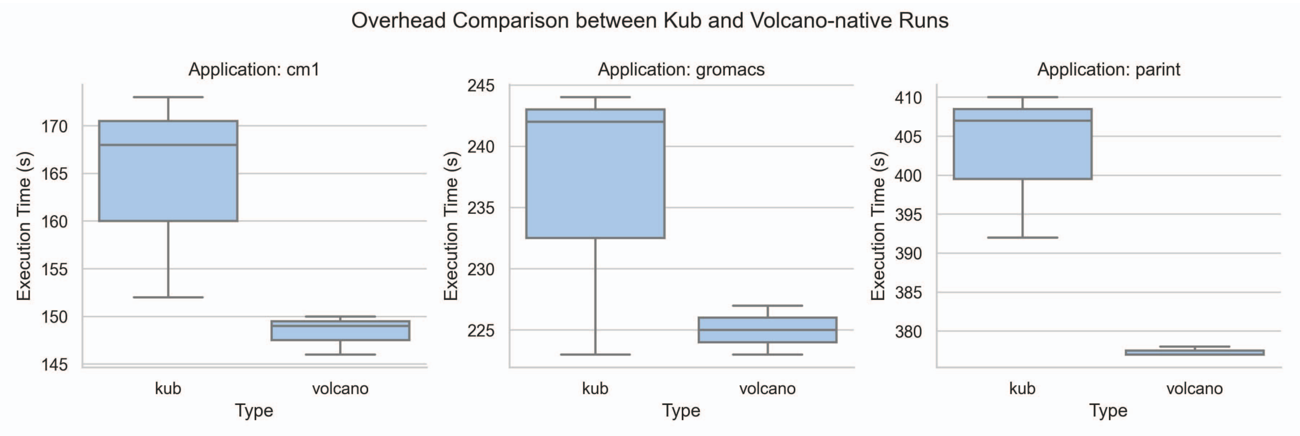


Fig. 3: Calculated overhead of the applications used in this work. Label *Kub* means that the applications were started using the custom launcher that coordinates the scaling, although no scaling was performed, while ”Volcano” is the traditional way of using MPI applications on Kubernetes. For this experiment, each application was executed using three MPI ranks, one per Kubernetes pod.

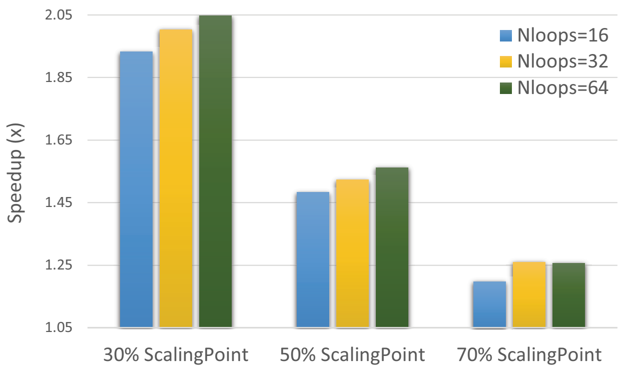


Fig. 4: Sensitivity test of increased compute intensity and benefit from scaling up from 2 to 6 ranks at three scaling points 30%, 50%, 70%, respectively.

B. Containerization of Applications

The process of building images for applications is widely documented, thus this paper will not discuss in-depth such process. We use Debian 11 “slim” as a base image for our containers, and build them using in two stages. The slim version of Debian removes many files related to documentation and language support, allowing the image to weigh roughly 80 MB (in comparison to the 125MB from the full image).

The two stages process consists in having the first image to compile the application itself with all the necessary building tools and development libraries. With that done, the compiled executable is transferred to the second image, which will contain only the necessary runtime libraries for execution.

C. Overhead of Kub

We evaluate the impact of using our approach (i.e., a launcher) in comparison to the vanilla version of the applications in Kubernetes. For this study, we did not perform any type of scaling as we wanted to measure the effects of i) the coordination among the pods and ii) the effect on thread-sharing of running a gRPC server and client in the system.

The timings were extracted from the total job duration from start (”Running” status) until completion (”Complete” status), and the boxplot in Figure 3 displays the results and also the conditions in which the application was executed.

The results show that there is a slight (between 10 to 15%) overhead for running *Kub* instead of the vanilla application in Kubernetes. However, this overhead can be largely attributed to the fact that the launcher sleeps between timesteps, so an action that should be performed during the time that the launcher is sleeping will be performed only at the beginning of the next timestep - which includes checking whether all executors are alive and/or even ending the execution for checkpointing.

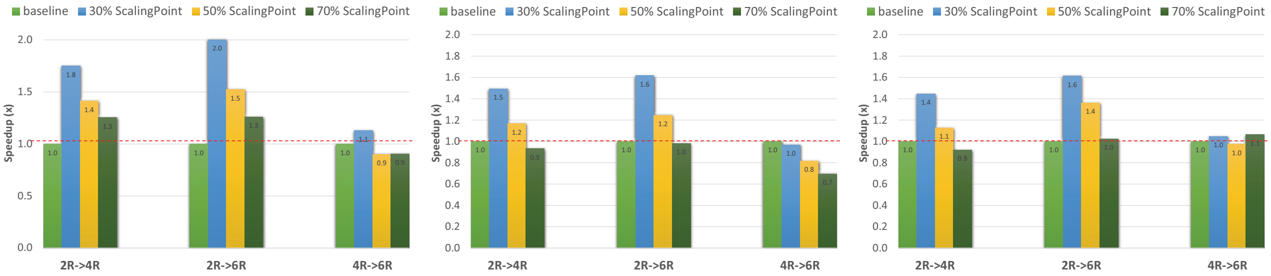
D. Scaling Experiments Overview

Sections V-E and V-F display results that are used for different analyses. However, all the experiments were designed and performed following similar procedures.

In particular, applications described in Section IV were used to study the effects of elastic scaling. As previously stated, the applications were not modified for these experiments; instead, an understanding of its checkpointing and resuming procedures was necessary and implemented into the launcher script.

In such experiments, the only factor when deciding to scale was time; there was no monitoring of available resources due to the hardware constraints - rather we assume that such resources will be available at a certain point of the execution of this application.

The speedup is calculated over a baseline time, which is a vanilla run of the application (i.e., without scaling). Each of



(a) PARINT

(b) CM1

(c) GROMACS

Fig. 5: The results for the elastic scaling performed by this work. Each case was executed 3 times, totalling 36 experiments per application. The label on the X axis refers to the *amount* of resources that is being introduced into the application, while the colours for each bar refer to *when* the scaling was performed. Refer to Sections V-F and V-G for an extensive discussion about this figure.

the experiments discussed above had their transition between ranks from one state to another at 30, 50 and 70% of the baseline time. This is done to investigate the influence of *when* the scaling is done. For the scenarios where the starting point is 2 MPI Ranks, the baseline is a full execution of 2 MPI Ranks; an analogue situation happens when the starting point is 4 MPI Ranks.

E. Sensitivity Test

This experiment uses PARINT to measure how applications with low or high computational intensity might benefit from the elastic scaling at different scaling points. We change the available NLOOPS parameter, where 16 makes the benchmark more memory-bound and 64 makes it more compute-bound. Figure 4 displays the obtained speedup over a baseline with no scaling. As the scaling point increases, the speedup decreases because there is less gain from scaling, but equal overhead from checkpointing. As for arithmetic intensity, the speedup increases as the intensity is increased showing that compute-bound tasks benefit more from parallelism than memory-bound tasks in our setup. Overall, a memory-bound task would have to scale out earlier in the execution to benefit, while a compute-bound task can scale out later in the execution.

F. Horizontal Scaling

In these experiments, we evaluate CM1, GROMACS and PARINT (with the NLOOPS parameter equaling 32) with our methodology, and the results can be seen in Figure 5.

For each application, we executed three different scenarios, all of them dealing with the increase of the available resources. The first scenario deals with an increase of 100% of resources (2 to 4 MPI ranks), the second one is a 200% increase (2 to 6 MPI ranks) and the last scenario is an increase of 50% (4 to 6 MPI ranks). For each of these scenarios, we evaluated the speedup at different moments of increasing the amount of resources. The relationship among the factors is mostly reconfirmed, and it is possible to observe one more relationship between the amount of resources and the speedup.

G. Discussions

There are two major insights to be drawn from the results shown in Sections V-C, V-E and V-F.

1) The decision of scaling or not depends on the amount of resources and how much time the application has expected to finish its execution.
2) Although there is overhead from using *Kub*, the possibility of scaling might enable the mitigation of it.

The first point is clearly illustrated by the results. In general, all three applications behave similarly: at the scenarios with the scaling point at 30% and 50% of the baseline time, there are improvements in the execution time for the case for scaling from 2 to 4 ranks and when scaling from 2 to 6 ranks, with the latter being usually faster than the former as the amount of resources is increased. Where there is scaling from 4 to 6 ranks, there is a perceived slowdown due to the increase of resources not being high enough to compensate for the time to stop for checkpointing and restarting.

However, this is not the case at 70% scaling point. Instead, there is a perceived slowdown on the application, meaning that if the application had run that much, it is better to let it finish instead of doing all the coordination for restarting.

That said, one question that arises is related to which applications are feasible to apply this methodology. As seen in Section V-E, PARINT is an ideal case and the speedup gains increase according to the arithmetical intensity of the application. In real applications, such as GROMACS and CM1, such gains are limited by the amount of non-computing operations (i.e., I/O) that are performed during its execution. Furthermore, such applications might also be able to stop and restart with a different number of ranks, also distributing the remaining load among the existing nodes.

Finally, in relation to the second point, the 10 to 15% of overhead that is shown between Volcano and *Kub* can be mitigated if a proper speedup is obtained with the resource increase - in particular, because the speedups increase ranges from 30 to 80%, as seen in Figure 5.

VI. Related Works

The emergence of converged cloud and HPC computing has attracted increased works in understanding the feasibility and gaps of scaling. We classified the literature related to this paper into four categories as follows.

Feasibility and infrastructure. Several works [3], [14], [21], [22] discussed the impact of using containers for HPC workloads, and evaluate orchestrators such as Docker Swarm and Kubernetes for such cases. In some cases, the latency impact of using InfiniBand and TCP/IP protocols is measured as well. Malleability is also proposed in MPI and PMIx [9]. Liu et al. [23] evaluated the impact of multi-tenancy in different types of containers (Docker and Singularity), considering both UMA and NUMA types of hardware, and reaching the conclusion that MPI applications suffer some degree of degradation due to each container being provided with its own networking namespace, with this effect being mitigated for applications that don't have much interprocess communication.

Malleability. There are several ongoing works in malleability for HPC applications. In particular, MPI Sessions was extended to support dynamic resource allocations [9]. Some parallel programming languages support a change in the number of nodes. In Charm++, for example, an interface named Converse Client Server sends and receives signals related to the expansion or reduction, and these signals can either be internal (the application takes its own decisions) or from an external application.

Scaling of HPC Workloads on the cloud. There is a trend of extending the Kubernetes scheduler to support HPC workloads better. In particular, Misale et al. [4] proposes a scheduler for Kubernetes called KubeFlux based on the ideas from Flux [24]. Using NFD, KubeFlux incorporates heterogeneous awareness for different compute resources. Milroy et al. [5] further contributed an MPI Operator and the Fluence plugin to Kubernetes, demonstrating scaling HPC applications up to 3000 MPI ranks on IBM Cloud and AWS.

Performance measurements and analysis. Gupta et al. [25] evaluated the performance and cost of selected HPC applications across multiple HPC and Cloud platforms. They focus on identifying suitable HPC workloads running on the cloud and proposed optimizations to Cloud virtualization mechanisms to match the characteristics of HPC workloads. Sukhija et al. [26] discussed the requirements of a monitoring tool in HPC environments and proposed the integration of a tool called OMNI (from NERSC) with current state-of-art tools that are used in cloud computing settings, such as Prometheus and Grafana.

VII. Conclusion

In this paper, we proposed a methodology for elastic scaling of tightly-coupled HPC workloads on the cloud. Our evaluation shows that the obtained speedup heavily relies on the quantity of resources to be introduced on the system and also at which point the scaling will be done, as there is a tradeoff between checkpointing overhead and the benefits of additional resources.

We show that the underlying mechanism for coordination between MPI processes on containerized environments is complex and deals with different technologies and software to turn the idea into reality - gRPC for coordination, Kubernetes for resource management, SSH and MPI for running tasks through the network, Volcano as a monitoring aid for the tasks, plus the application-level knowledge for ensuring that the checkpointing works. Furthermore, our work can be advanced in two future fronts of work.

The first front is monitoring (Section II-E). The current work does not leverage resource awareness but rather builds a fixed time model for simplicity (i.e., at 30%, 50% and 70% of a base execution time). For the elastic scaling to be effective in a production-level system, two factors should be considered: i) how many resources will be available on the system to an application, and ii) how much can the application benefit from additional resources. This is widely studied in cloud environments, especially when dealing with quality of service for users.

The second front is to analyse more complex patterns of scaling. This work only scaled up once although *Kub* can do it multiple times through the same algorithm. However, designing experiments and analyzing results for such patterns is difficult as there is a large space to explore. Finally, as one can scale up for performance, we think that scaling down can also play a big role in resource management and energy consumption in the future.

Acknowledgments

This research is supported by the European Commission under the Horizon project OpenCUBE (GA-101092984).

References

[1] J. Ejarque, R. M. Badia, L. Albertin, G. Aloisio, E. Baglione, Y. Becerra, S. Boschert, J. R. Berlin, A. D'Anca, D. Elia *et al.*, "Enabling dynamic and intelligent workflows for HPC, data analytics, and AI convergence," *Future generation computer systems*, vol. 134, pp. 414–429, 2022.

[2] D. Medeiros, G. Schieffer, J. Wahlgren, and I. Peng, "A GPU-Accelerated Molecular Docking Workflow with Kubernetes and Apache Airflow," in *International Conference on High Performance Computing*. Springer, 2023, pp. 193–206.

[3] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant, "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)*. IEEE, 2019, pp. 11–20.

[4] C. Misale, D. J. Milroy *et al.*, "Towards standard Kubernetes scheduling interfaces for converged computing," in *Smoky Mountains Computational Sciences and Engineering Conference*. Springer, 2021, pp. 310–326.

[5] D. J. Milroy, C. Misale, G. Georgakoudis, T. Elengikal, A. Sarkar, M. Drocco, T. Patki, J.-S. Yeom, C. E. A. Gutierrez, D. H. Ahn *et al.*, "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, 2022, pp. 57–70.

[6] D. Araújo De Medeiros, S. Markidis, and I. Peng, “LibCOS: Enabling Converged HPC and Cloud Data Stores with MPI,” in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2023, pp. 106–116.

[7] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, “Geo-distributed efficient deployment of containers with Kubernetes,” *Computer Communications*, vol. 159, pp. 161–174, 2020.

[8] N. R. Herbst, S. Kounev, and R. H. Reussner, “Elasticity in Cloud Computing: What It Is, and What It Is Not.” in *ICAC*, vol. 13, no. 2013, 2013, pp. 23–27.

[9] D. Huber, M. Streubel, I. Comprés, M. Schulz, M. Schreiber, and H. Pritchard, “Towards Dynamic Resource Management with MPI Sessions and PMIx,” in *Proceedings of the 29th European MPI Users' Group Meeting*, 2022, pp. 57–67.

[10] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong, “The Characteristics of Cloud Computing,” in *2010 39th International Conference on Parallel Processing Workshops*, 2010, pp. 275–279.

[11] W. D. Mulia, N. Sehgal, S. Sohoni, J. M. Acken, C. L. Stanberry, and D. J. Fritz, “Cloud workload characterization,” *IETE Technical Review*, vol. 30, no. 5, pp. 382–397, 2013.

[12] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute,” *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017.

[13] H. Gantikow, S. Walter, and C. Reich, “Rootless Containers with Podman for HPC,” in *High Performance Computing: ISC High Performance 2020 International Workshops*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 343–354.

[14] N. Marathe, A. Gandhi, and J. M. Shah, “Docker swarm and kubernetes in cloud computing environment,” in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE, 2019, pp. 179–184.

[15] R. H. Castain, T. S. Woodall *et al.*, “The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing,” in *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[16] N. Tolaram, “cadvisor,” in *Software Development with Go: Cloud-Native Programming using Golang with Linux and Docker*. Springer, 2022, pp. 347–376.

[17] G. H. Bryan and J. M. Fritsch, “A benchmark simulation for moist nonhydrostatic numerical models,” *Monthly Weather Review*, vol. 130, no. 12, pp. 2917 – 2928, 2002. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/130/12/1520-0493_2002_130_2917_absfmn_2.0.co_2.xml

[18] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, “GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers,” *SoftwareX*, vol. 1, pp. 19–25, 2015.

[19] C. Kutzner *et al.*, “GROMACS in the Cloud: A Global Supercomputer to Speed Up Alchemical Drug Design,” *Journal of Chemical Information and Modeling*, vol. 62, no. 7, pp. 1691–1711, 2022, pMID: 35353508. [Online]. Available: https://doi.org/10.1021/acs.jcim.2c00044

[20] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, “Applying the roofline model,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 76–85.

[21] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, “Evaluation of docker containers for scientific workloads in the cloud,” in *Proceedings of the Practice and Experience on Advanced Research Computing*, 2018, pp. 1–8.

[22] S. Abraham, A. K. Paul, R. I. S. Khan, and A. R. Butt, “On the use of containers in high performance computing environments,” in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 284–293.

[23] P. Liu and J. Guitart, “Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study,” *The Journal of Supercomputing*, vol. 77, pp. 6273–6312, 2021.

[24] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland *et al.*, “Flux: Overcoming scheduling challenges for exascale workflows,” *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020.

[25] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B.-S. Lee, V. March, D. Milojicic, and C. H. Suen, “Evaluating and improving the performance and scheduling of HPC applications in cloud,” *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 307–321, 2014.

[26] N. Sukhija and E. Bautista, “Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus,” in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 2019, pp. 257–262.