Virtual Clusters: Isolated, Containerized HPC Environments in Kubernetes

George Zervas[1,2], Antony Chazapis[1], Yannis Sfakianakis[1,2], Christos Kozanitis[1], and Angelos Bilas[1,2]

[1] Institute of Computer Science, FORTH
[2] Computer Science Department, University of Crete
{georgzerb, chazapis, jsfakian, kozanitis, bilas}@ics.forth.gr

Abstract. Today, Cloud and HPC workloads tend to use different approaches for managing resources. However, as more and more applications require a mixture of both high-performance and data processing computation, convergence of Cloud and HPC resource management is becoming a necessity. Cloud-oriented resource management strives to share physical resources across applications to improve infrastructure efficiency. On the other hand, the HPC community prefers to rely on job queueing mechanisms to coordinate among tasks, favoring dedicated use of physical resources by each application.
In this paper, we design a combined Slurm-Kubernetes system that is able to run unmodified HPC workloads under Kubernetes, alongside other, non-HPC applications. First, we containerize the whole HPC execution environment into a *virtual cluster*, giving each user a private HPC context, with common libraries and utilities built-in, like the Slurm job scheduler. Second, we design a custom Slurm-Kubernetes protocol that allows Slurm to dynamically request resources from Kubernetes. Essentially, in our system the Slurm controller delegates placement and scheduling decisions to Kubernetes, thus establishing a centralized resource management endpoint for all available resources. Third, our custom Kubernetes scheduler applies different placement policies depending on the workload type. We evaluate the performance of our system compared to a native Slurm-based HPC cluster and demonstrate its ability to allow the joint execution of applications with seemingly conflicting requirements on the same infrastructure with minimal interference.

Keywords: Cloud-native HPC · Kubernetes scheduling · Slurm

1 Introduction

Cloud and HPC computing environments are mostly similar in hardware specifications, but differ largely in the software stack and how it manages available resources. Cloud providers use virtualization mechanisms to facilitate sharing, valuing colocation of workloads to the point of overprovisioning, whereas in HPC clusters workloads are allocated exclusive resources, based on exact requirements given by the user when submitting the respective job. As the complexity of modern applications increases, it is not uncommon for deployments to include parallel provisioning of backend services, as well as on-demand execution of data analytics pipelines and HPC codes. For such workloads, it is essential to accommodate both resource allocation schemes on the same hardware infrastructure, exploiting resource sharing, but also avoiding interference as much as possible.

In this paper, we explore the convergence of Cloud and HPC in a common, container-based environment, backed by Kubernetes, the most prominent distributed container orchestration framework [3]. Containers are gaining ground as the preferred deployment method in the Cloud, as they implement a convenient packaging scheme for applications, they are lightweight when running, and provide isolation between instances for security purposes. Kubernetes provides abstractions for hardware resources and automatically scales service replicas to meet demand, while keeping redundancies to alleviate for unadvertised failures. The HPC world has cautiously been following the trend, primarily utilizing containers as a portable method to bundle applications with associated library dependencies. These containers are then typically submitted as jobs using Slurm, a popular workload manager responsible for coordinating the allocation of resources throughout the cluster, via shared submission queues.

To run HPC applications in Kubernetes, we introduce the concept of the *virtual cluster*, as a group of multiple container instances that function as a unified cluster environment from the user's perspective. Each node in a virtual cluster embeds all necessary libraries and utilities, as well as a private Slurm deployment; the user working inside a virtual cluster can only view and manage jobs submitted from within the same context. In practice though, each such Slurm setup is not independent. We extend the Slurm controller with a custom protocol, to communicate with the central Kubernetes scheduler when requiring resources, effectively placing Kubernetes in charge of resource allocations for the whole cluster. Moreover, we use *Genisys*, a custom Kubernetes scheduler that distinguishes between "HPC" and "data center" services (typical Kubernetes deployments that run in other containers), in order to apply different allocation policies and maximize overall usage. In cases where HPC workloads do not consume all node-local resources, Genisys colocates data center services, while constantly satisfying their user-defined performance targets. Therefore, HPC and data center workloads execute transparently on the same infrastructure, achieving high levels of CPU utilization.

This integration has several benefits: (i) Compatibility: Supporting Slurm inside virtual clusters is crucial in order to keep compatibility with existing Slurm scripts. (ii) Colocation: By containerizing the whole runtime environment and using Kubernetes as the substrate, we are able to run hybrid workloads on top of the same physical cluster, optimizing for high utilization and avoiding static cluster partitioning for HPC and data center tasks. (iii) Portability: The containerized environment offered with virtual clusters allows users to install different dependencies without polluting the bare metal infrastructure and avoid issues with conflicting versions of the same libraries. It also makes migrating to a different Kubernetes cluster possible, just by transferring the container images to the other system and deploying them using the same Kubernetes objects.

2 Related Work

The integration of HPC job management in the context of Kubernetes has been addressed in several studies. In [13], the authors use a utility called *hpc-connector* that acts as an HPC job proxy: Users submit respective Kubernetes jobs with the appropriate settings, and hpc-connector forwards them to the HPC cluster, monitors their execution, and collects their results. This solution can probably be used with containers to address portability issues. On the other hand, the main focus is on HPC job management with a Kubernetes-compatible interface; the HPC and cloud clusters are treated as two separate environments making it impossible to monitor and place cloud and HPC workloads over the same physical cluster. A similar approach is presented in [20], where a Kubernetes installation is interfaced to a Torque-based HPC cluster.

One critical aspect of the containerization of HPC workloads is runtime performance when compared to an actual physical cluster. Related work has measured the network performance of containerized HPC codes, and findings suggest that there is little to no performance overhead when an InfiniBand network is used [5]. In general, Docker containers do not introduce significant performance overheads, while in some cases they can provide better QoS due to the usage of cgroups resource limiting mechanisms when compared with a bare metal runtime environment [8–10].

There is a plethora of papers that handle the scheduling of workloads with the main goal of increasing the utilization in the infrastructure. Sparrow [16] and Eagle [7], handle the scheduling of application tasks in clusters. Sparrow focuses on speed, but can not handle workloads with conflicting goals as in our case. Eagle follows a hybrid approach, with a centralized component that performs careful placement of long-running tasks, and a distributed component emphasizing on quick placement. Our scheme is also hybrid, however, with different goals. Ursa [11] is a task scheduler for spark-monotasks [15] and Rhythm [19] is a data center scheduler that ensures the latency of latency-critical applications. Both works colocate "compatible" tasks to increase the utilization in the underlying infrastructure, but do not effectively guard against interference. In contrast, our approach constantly monitors application performance and adjusts container placement and resource allocations at runtime to achieve a user-defined performance target. Control loops for dynamically adjusting resources based on runtime performance have been used in systems such as SLAOrchestrator [14] and Skynet [17]. The former tries to optimize cost of services when running in the Cloud, while the latter optimizes hardware efficiency by colocating more applications on the same nodes, as long as they acheive their user-defined performance targets. Genisys's handling of data center tasks is based on Skynet, extended to allocate HPC tasks with different, placement-based constraints.

3 Design Overview

A *virtual cluster* is a group of container instances that virtualizes an environment to run HPC workloads that use MPI and other software frameworks. From the perspective of applications, virtual clusters are indistinguishable from physical nodes that execute instances of MPI processes in parallel, as all physical processing cores, RAM, the low-latency InfiniBand network, and accelerators are available in each container context. Each virtual cluster spans all physical nodes and multiple virtual clusters can co-exist over the same set of physical nodes, as shown in Fig. 1, which presents the high-level design concept.

Inside each virtual cluster, as part of the bundled software stack, we deploy a private Slurm context, so users can invoke existing scripts to run HPC workloads. One of the virtual cluster nodes acts as the Slurm controller, while all virtual cluster nodes run the Slurm agent and register with the controller. Configuration of the Slurm deployment is automatically done at virtual cluster initialization. Unmodified, the virtual-cluster-local Slurm would perform resource allocation and scheduling of Slurm jobs as if it were in control of the whole cluster. Each independent Slurm installation is isolated inside its own containers and does not account for the presence of other containers running and consuming computing resources; that being other virtual clusters or typical Kubernetes services.

To schedule and place workloads across multiple virtual clusters and prevent the interference introduced by overlapping jobs, we have modified the Slurm controller's placement mechanism to delegate all respective decisions to the external Kubernetes scheduler. The Kubernetes scheduler in this scheme is the central authority that has the full knowledge of the cluster's current resource allocations and acts as a global coordinator for new requests. Moreover, Genisys, our custom Kubernetes scheduler implementation (described in Section 4) distinguishes between "HPC" and "data center" type workloads, in order to improve the overall utilization of available hardware. Data center services do not use virtual clusters, but are deployed in Kubernetes as deployments, jobs, or other API objects that execute in containers running alongside the ones used by virtual clusters.

Genisys ensures each virtual cluster does not share resources with other virtual clusters or data center services. When a new service is deployed, Genisys iterates over an internal free resource list for each node and attempts to find which nodes can accommodate it. If *resource oversubscription* is not enabled Genisys will always place tasks on nodes with enough free resources to fit in. Furthermore, the scheduler supports two different placement policies: (i) The *Least Loaded Selection Policy* attempts to place tasks to the least loaded nodes on the cluster, effectively spreading out the load, allowing to fit more jobs on a given set of nodes to achieve higher cluster resource utilization. (ii) The *Max Loaded Selection Policy* attempts to pack as many services in nodes, allowing for higher energy efficiency, with the danger of not being able to fit as many tasks on the cluster (as some "loaded" nodes will not have enough free resources).

For data center workloads, Genisys allows sharing of resources, by estimating the aggregate resources that are required to achieve a user-defined performance objective (i.e., latency, throughput). It manages four types of resources: number of cores, memory size, I/O bandwidth, and network bandwidth. Genisys performs its estimations using a feedback control loop similar to Skynet. Afterwards, it decides on the size, the number, and the placement of containers in physical nodes according to the selected policy.

Colocating HPC tasks with the data center services is configurable. The default behaviour allows tasks of both types to use the same nodes and share resources. The other option is to perform type-based placement, implicitly partitioning the nodes by placing HPC tasks on some nodes and data center tasks on others. This approach may minimize interference introduced by task colocation, but also reduces efficiency, and is not used in this paper.

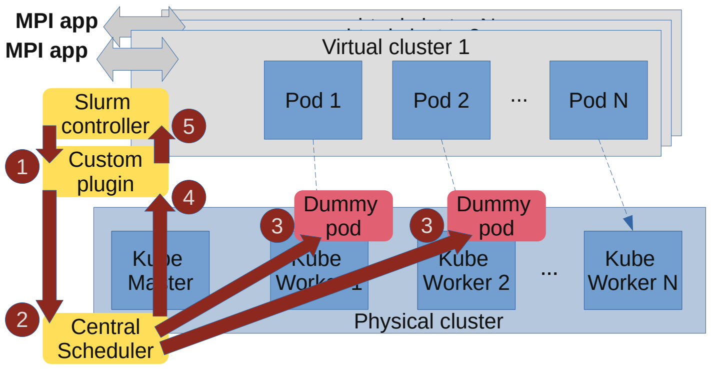

Fig. 1. Each container instance of a virtual cluster runs in a different physical machine, while multiple virtual clusters may run in parallel. The custom Slurm job placement plugin communicates with Genisys to perform job placement. These jobs are visible at the Kubernetes level as "dummy allocations".

Fig. 1 illustrates the steps involved in the communication between virtual clusters and the cluster-wide Kubernetes scheduler for HPC workloads: 1. On job initialization Slurm sends an allocation request to the main Kubernetes controller via a custom plugin. In this request Slurm specifies the resources needed for the job (node count, CPU count, etc.). 2. The custom plugin forwards these allocations to Genisys, by allocating "dummy pods" in Kubernetes, with resource requirements matching the Slurm job's specifications. Dummy containers are practically idle; they consume no resources themselves, but act as placeholders for the allocation of resources that will be used by the actual jobs inside the virtual cluster. 3. On receiving a dummy allocation for a Slurm job placement, Genisys iterates over the available nodes and checks if enough resources are available. If so, it schedules the dummy containers for execution. 4. The allocation is communicated back to the custom Slurm plugin as a node list. 5. The plugin, in turn, forwards the response to the Slurm controller. The node list contains the selected nodes for the Slurm job deployment. If no suitable nodes are found, the controller puts the job on hold.

4 Implementation

Preparing and deploying virtual clusters: Virtual cluster container images are prepared as "typical" Docker images, by starting from some base Linux distribution and adding layer after layer of development tools, libraries, and other software. Our reference images are based on CentOS and the Mellanox OpenFabrics Enterprise Distribution (OFED), which includes Open MPI with InfiniBand support as well as other libraries. In addition, we install several extra libraries and frameworks (i.e., CUDA, GROMACS, TensorFlow, Horovod), Slurm, as well as utilities to help in evaluating application performance. This base container recipe is available to our users, so they can tailor it to their needs, using different software versions or supplementary libraries and tools.

Upon instantiation, each virtual cluster container actually runs the SSH daemon as its primary process. The instance startup script first waits for all pods (nodes in the virtual cluster) to be ready and then creates all necessary configuration: keys for password-less SSH connectivity, MPI hostfile, and Slurm configuration at `/etc/slurm.conf`. As the last step, it starts Slurm (the Slurm controller runs in the first container). Each virtual cluster is deployed using a Kubernetes DaemonSet, which assigns one pod per physical machine. As HPC application developers usually assume similar capabilities and equal network-level distances across nodes, placing a single pod in each node is convenient and produces expected results.

Slurm-Kubernetes interface: The Slurm controller uses a node bitmap in order to represent resource reservations in available HPC nodes and find candidates to place incoming jobs. For placement, Slurm uses the `_job_test()` function, which is called by the controller when a new job arrives. `_job_test()` receives the job's resource requirements and the node bitmap. It then checks if a set of computing nodes is available by calling `_select_nodes()`, which returns a list of selected nodes, so Slurm can proceed to mark them in the node bitmap and start the job. If the selection process returns an empty node list, then the job is rescheduled for later placement. To delegate all job placement decisions externally to Kubernetes, we first ignore the node bitmap returned by `_select_nodes()`. Instead, we implement a custom plugin written in Golang that receives the job's resource requirements (CPUs, RAM, node count) and creates a mirror Kubernetes allocation of "dummy pods" using the same resources. The plugin runs next to each Slurm controller communicating with Kubernetes via the API server. When the dummy pods are placed, we return the node list for the specified job back to the Slurm controller. On receiving the list, we trigger Slurm to modify the node bitmap and place the job.

Scheduling extensions: The scheduling of dummy pods does not *require* a custom Kubernetes scheduler. However without special arrangements for HPC workloads, the default scheduler may place multiple HPC jobs on the same nodes, maximizing interference. To this end, we have extended our Genisys scheduler to support both "HPC" and "data center" workloads and enforce different types of placement policies. We label HPC workloads as "SLURM-JOB", in order to distinguish them from other workloads running on the same cluster. Allocations for Slurm applications happen in a static manner and Genisys can be configured to avoid colocating them with other jobs marked as “SLURM-JOB” for optimal performance. This policy may be selected because of the lack of available metrics offered by MPI applications and their sensitivity due to synchronization barriers.

On the other hand, data center workloads include a user-defined performance objective, which Genisys must achieve during their execution. Genisys monitors periodically the performance of each running data center service to get feedback about the effectiveness of its current resource allocation, using the Kubernetes Custom Metric Server and Prometheus [1]. In case of a performance violation in a workload, Genisys increases its resource allocation according to the measured drop in performance and vice versa. For new resource estimations, Genisys also considers the history of previous performance measurements, which is affected mainly by the workload mix.

5 Evaluation

We evaluate our system by running a mixture of MPI workloads and other services on the same cluster, and measuring the overall efficiency through the total runtime of all applications combined and the individual runtime per application. Our hardware setup consists of 5 servers, each with a single 32-core/64-thread AMD EPYC 7551P processor (running at 2.00GHz) and 128GB of memory, for 320 threads in total. All servers have SSD storage devices and are interconnected via 56Gb/s InfiniBand. We run Kubernetes 1.19.7 on CentOS 7.6.

We have created a multi step MPI workload using benchmarks from the NAS Parallel Benchmark Suite [4]. We choose realistic HPC task sizes and their distribution by following traces outlined in [18], which analyzes the HPC workloads run on the Lomonosov-2 supercomputer, categorized according to resource allocation sizes and CPU consumption. We allocate 5% of the workload’s CPU time to 16 thread (small), 20% to 32 thread (medium), 65% to 64 thread (medium-large), and 10% to 128 thread (large) jobs. The “data center” workload consists of: (i) 5 Nginx servers offering static content, each allocating 6 CPU threads. The servers are hit with 200,000 total requests from Apache Bench [2]. (ii) 5 memcached servers, each allocating 6 CPU threads. The servers are hit with a 200 million operation workload generated by YCSB [6]. (iii) Spark benchmarks from the Spark-Bench [12] performance suite, using 5 Spark workers, each allocating 20 CPU threads as a Kubernetes job.

As a baseline, we first run the full HPC workload on all nodes exclusively and then the data center tasks, representing the typical scenario where Slurm and Kubernetes time-share the same resources. For the other scenarios, we deploy both workloads concurrently, colocating them over the same physical resources, in 6 different configurations: (i) *Genisys Least Loaded Policy:* We use the Genisys scheduler and our modified version of Slurm that communicates with Genisys for placement decisions. In Genisys we select the *Least Loaded Policy.* (ii) *Genisys Max Loaded Policy:* Same as the previous configuration, but using Genisys’s *Max Loaded Policy.* (iii) *Unmanaged:* We use the default Kubernetes scheduler and unmodified Slurm in virtual clusters. (iv) *Partitioned:* Like Unmanaged, but we statically partition the nodes into a 2-node Kubernetes and a 3-node Slurm cluster. (v) *Thread Partitioned 50%:* Like Unmanaged, but we partition the cluster's nodes by giving 50% of each node's CPU capacity to Kubernetes and the rest 50% to Slurm. Kubernetes and Slurm are configured to each use half of the CPU resources of each node. (vi) *Thread Partitioned 75%:* Same as the previous configuration, but by giving 75% of each node's CPU capacity to Kubernetes and 75% to Slurm. The HPC and data center workloads partially overlap over the same physical resources, as Slurm and Kubernetes see a total of (150%) of each node's capacity available.

The execution time of each workload step for the different scenarios is shown in Fig. 2. In the first graph we show the individual execution times of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization over time for each different scenario.

Effect of different policies in Genisys: In general, the Least Loaded scenario is able to run more tasks in parallel and achieve higher cluster utilization, as spreading the tasks to the least loaded nodes allows for more efficient fitting when compared to choosing the most loaded nodes. In the case of the Max Loaded scenario, filling the most loaded nodes first, often leads to situations where tasks that request a specific number of nodes cannot fit into the cluster. Some nodes of the cluster are fully loaded and the number of nodes with enough space is smaller than the requested number of nodes. The Least Loaded policy achieved (14%) lower total execution times when compared to Max Loaded. There was also a (4%) higher individual task performance. In the next sections

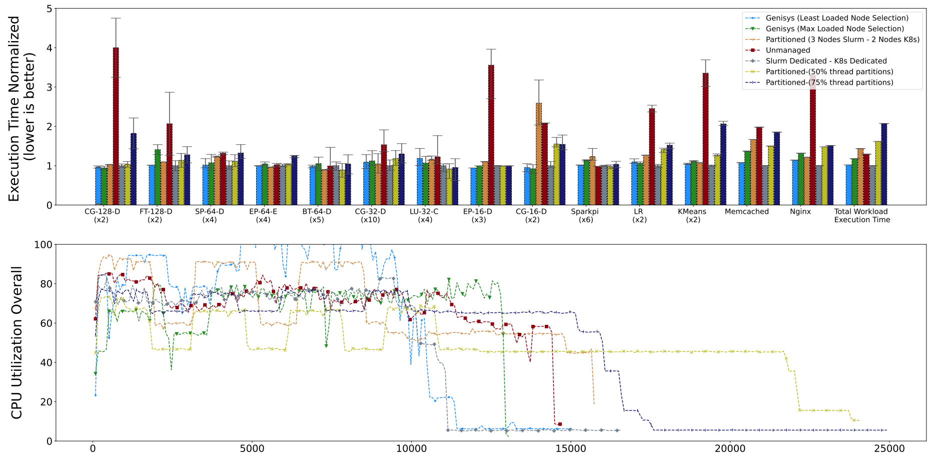

Fig. 2. Performance comparison of the different configurations

we use the Least Loaded Policy in order to evaluate Genisys in contrast to other configurations.

Genisys vs Unmanaged: The Unmanaged setup introduces high interference between the tasks as Kubernetes and Slurm are not able to coordinate placement, which leads to resource over-subscription (both Kubernetes and Slurm see each individual node's resources as 100% available). Especially in the case of MPI tasks, this has catastrophic results in their performance, as threads running in congested nodes will slow down the whole job. During the runs, we observe that Slurm selects nodes serially, so jobs are packed in the first cluster nodes, which leaves other nodes underutilized. The Kubernetes default scheduler, on the other hand, places tasks in a round-robin fashion. The total execution time needed by Genisys's Least Loaded Policy to complete the combined workload is 11200 seconds, which is 25% faster when compared to Unmanaged (14633 seconds). On average the individual execution times are 28% faster when using Genisys compared to Unmanaged. Also, Genisys achieves higher average CPU utilization (90%), compared to Unmanaged (71%).

Genisys vs Partitioned: The Partitioned approach allows both HPC and data center tasks to have optimal performance as there is no resource overlapping, sacrificing, however, overall utilization. Due to restricting workloads into their corresponding partition, Slurm can not leverage resources from the Kubernetes cluster even when the execution of the data center tasks finishes. This results in a higher total workload execution time of 15800 seconds, 34% slower when compared to Genisys (11200 seconds). The average individual task completion time is 4% lower in the Genisys case. We assume that this is because Genisys spreads the tasks across all the cluster nodes and is able to better utilize the RDMA network. Again, Genisys achieves higher average CPU utilization (90%) compared to Partitioned (75%).

Genisys vs Thread Partitioned: While partitioning the cluster at the CPU level is an uncommon approach, it is interesting to compare it to Genisys, as Genisys allows resource sharing between data center and HPC tasks over the same physical nodes in a similar manner. The main goal of this experiment is to show that with Genisys we are going to have better resource utilization as workloads are not restricted to their respective partitions, so when one partition is underutilized the other will be able to leverage the free resources. The Thread Partitioned 50% scenario results in higher total workload execution time, 44% slower when compared to Genisys. The average individual task completion time is 4.5% lower in the Genisys case. Genisys achieves higher average CPU utilization (82%), compared to Thread Partitioned 50% (56%). In the Thread Partitioned case, when the data center workload finishes, Slurm is not able to utilize the Kubernetes nodes. Also due to the smaller number of threads available to both Slurm and Kubernetes, the tasks can not fit as efficiently as when running with Genisys. Similar results are obtained when overprovisioning, by assigning 75% of the resources to each partition. In Thread Partitioned 75%, the average individual task completion time is 28% lower in the Genisys case, which we attribute to even higher interference in congested nodes.

6 Conclusion

This paper presents a method to run HPC workloads in Kubernetes using portable and extensible containerized environments called *virtual clusters*. Virtual clusters include Slurm, so users can run existing scripts unmodified, and are deployed alongside other Kubernetes services on the same physical nodes. Without any additional changes, the resulting hybrid resource allocation environment would have individual Slurm instances operating within their virtual cluster constraints, unaware of what is happening at the overall cluster-level, where decisions are made by Kubernetes. To avoid resource allocation conflicts, we integrate Slurm with Kubernetes, by extending the Slurm controller to delegate placement decisions to Genisys, our custom Kubernetes scheduler.

Our evaluation results indicate that it is not only possible to colocate data center tasks with HPC jobs when remaining CPU cycles are available, but with appropriate scheduling it can be beneficial to overall performance. Overall, Genisys is able to integrate Slurm into the Kubernetes ecosystem with minimal performance overhead across different task categories. The evaluation shows that with the use of Genisys it is possible to reduce the execution time of combined workloads compared to unmanaged and partitioned approaches.

Acknowledgements. We thankfully acknowledge the support of the European Commission under the Horizon 2020 Programme through projects HiPEAC (GA-871174) and EVOLVE (GA-825061).

References

1. An open-source monitoring solution. https://prometheus.io/
2. The apache software foundation. apache http server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html
3. VMware: The State of Kubernetes 2020. https://k8s.vmware.com/state-of-kubernetes-2020/
4. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The nas parallel benchmarks. Int. J. High Perform. Comput. Appl. **5**(3), 63–73 (Sep 1991)
5. Beltre, A.M., Saha, P., Govindaraju, M., Younge, A., Grant, R.E.: 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). pp. 11–20 (2019)
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing. p. 143–154. SoCC '10, ACM, New York, NY, USA (2010)
7. Delgado, P., Didona, D., Dinu, F., Zwaenepoel, W.: Job-aware scheduling in eagle: Divide and stick to your probes. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 497–509. SoCC '16, ACM, New York, NY, USA (2016)

8. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 171–172 (2015)
9. Herbein, S., Dusia, A., Landwehr, A., Mcdaniel, S., Monsalve Diaz, J.M., Yang, Y., Seelam, S., Taufer, M.: Resource management for running hpc applications in container clouds. pp. 261–278 (06 2016)
10. Higgins, J., Holmes, V., Venters, C.: Orchestrating docker containers in the hpc environment. pp. 506–513 (07 2015)
11. Jin, T., Cai, Z., Li, B., Zheng, C., Jiang, G., Cheng, J.: Improving resource utilization by timely fine-grained scheduling. In: Proceedings of the Fifteenth European Conference on Computer Systems. pp. 1–16 (2020)
12. Li, M., Tan, J., Wang, Y., Zhang, L., Salapura, V.: Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In: Proceedings of the 12th ACM International Conference on Computing Frontiers. CF '15, ACM, New York, NY, USA (2015)
13. López-Huguet, S., Segrelles, J.D., Kasztelnik, M., Bubak, M., Blanquer, I.: Seamlessly managing hpc workloads through kubernetes. In: Jagode, H., Anzt, H., Juckeland, G., Ltaief, H. (eds.) High Performance Computing. pp. 310–320. Springer International Publishing, Cham (2020)
14. Ortiz, J., Lee, B., Balazinska, M., Gehrke, J., Hellerstein, J.L.: Slaorchestrator: Reducing the cost of performance slas for cloud data analytics. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 547–560. USENIX Association, Boston, MA (Jul 2018)
15. Ousterhout, K., Canel, C., Ratnasamy, S., Shenker, S.: Monotasks: Architecting for performance clarity in data analytics frameworks. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 184–200 (2017)
16. Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I.: Sparrow: distributed, low latency scheduling. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 69–84. ACM (2013)
17. Sfakianakis, Y., Marazakis, M., Bilas, A.: Skynet: Performance-driven resource management for dynamic workloads. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). IEEE (2021)
18. Shvets, P., Voevodin, V., Nikitenko, D.: Approach to Workload Analysis of Large HPC Centers, pp. 16–30 (07 2020)
19. Zhao, L., Yang, Y., Zhang, K., Zhou, X., Qiu, T., Li, K., Bao, Y.: Rhythm: component-distinguishable workload deployment in datacenters. In: Proceedings of the Fifteenth European Conference on Computer Systems. pp. 1–17 (2020)
20. Zhou, N., Georgiou, Y., Zhong, L., Zhou, H., Pospieszny, M.: Container orchestration on hpc systems. In: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). pp. 34–36 (2020)