sciendo

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **22**, No 3
Sofia • 2022 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2022-0029

Serverless High-Performance Computing over Cloud

Davit Petrosyan, Hrachya Astsatryan

Institute for Informatics and Automation Problems of National Academy of Sciences of the Republic of Armenia
E-mails: davit.petrosyan@iiap.sci.am hrach@sci.am

Abstract: *HPC clouds may provide fast access to fully configurable and dynamically scalable virtualized HPC clusters to address the complex and challenging computation and storage-intensive requirements. The complex environmental, software, and hardware requirements and dependencies on such systems make it challenging to carry out our large-scale simulations, prediction systems, and other data and compute-intensive workloads over the cloud. The article aims to present an architecture that enables HPC workloads to be serverless over the cloud* (*Shoc*), *one of the most critical cloud capabilities for HPC workloads. On one hand, Shoc utilizes the abstraction power of container technologies like Singularity and Docker, combined with the scheduling and resource management capabilities of Kubernetes. On the other hand, Shoc allows running any CPU-intensive and data-intensive workloads in the cloud without needing to manage HPC infrastructure, complex software, and hardware environment deployments.*

Keywords: *HPC, MPI, Kubernetes, containerization, cloud.*

AMS subject classifications: 68T09.

1. Introduction

High-Performance Computing (HPC) infrastructures address compute-intensive simulations delivering results in a reasonable time [1]. Various combinations of parallel programming paradigms entirely harvest the computational capabilities of heterogeneous architectures of HPC systems by increasing performance and energy efficiency. Open Multi-Processing (OpenMP) is a popular shared-memory system utilizing non-scalable synchronization mechanisms and controlling data distribution within HPC systems' nodes [2]. Message Passing Interface (MPI) is a well-known inter-node programming model dealing with parallelization aspects, such as data and task distribution or communication and synchronization [3]. MPI+X is used for the hybrid model, where X represents the intra-node programming model.

With the rapid increase in make use of cloud computing systems, HPC clouds are becoming an alternative to bare-metal solutions by providing threads at the Operating System (OS) level and Virtual Machines (VM) at the hypervisor level [4].

HPC clouds increase the utilization of computational resources by delivering dynamic and on-demand reallocation of resources among applications and users. The HPC cloud platforms may provide access also accelerator based HPC resources, mainly General-Purpose computation on Graphical Processing Units (GPGPU) using Compute Unified Device Architecture or Open Computing Language programming models [5]. Another opportunity of HPC clouds is to provide HPC as a service for Big Data processing applications, such as Apache Hadoop or Spark, together with MapReduce [6]. Last decade the demand for MPI-based and GPGPU-based HPC applications to use cloud-based environments has been increasing [7].

The HPC cloud computing platform is closely related to middleware technology. The OpenStack [8] open-source cloud middleware delivers the Infrastructure as a Service (IaaS) platform through different virtualized services such as CPU or storage virtualization. Although, the Docker container provides easy-to-manage and cost-effective microservices without launching entire VMs. Instead of VMs, multiple isolated containers share the host operating system and physical resources, impacting performance, memory requirement, and infrastructure cost [9]. Docker Swarm and Kubernetes [10] are the most well-known container-based virtualization technologies focusing specifically on cluster-based systems. These technologies provide extensive functionalities, such as system administration, resource management schemes, security policy enforcement, or network access.

The complexity of HPC workflows is a limitation for the provisioning of HPC cloud services based on shared memory, distributed memory, or Big Data processing programming models, such as workload management, environment isolation, scalability, or efficiency. The heterogeneous software and hardware resources are additional barriers to managing large-scale HPC infrastructures comprising multiple parallel computing platforms. The workload orchestration requires developing advanced methods, services, and algorithms, such as resource management and scheduling system to control tasks and allocate resources within given requirements and availability. However, this is more critical when resources are limited, and workflows are resource-intensive. It is challenging to manage all this together without proper infrastructure and requires lots of manual effort to set this up, even on a cluster with few nodes. Therefore, the main limitations of the suggested methods include:

- The complexity of deployment implementations,
- End-user awareness of multiple technologies,
- The usage of scheduling sub-systems impacting the performance.

The paper presents a Kubernetes-based HPC cloud architecture called Shoc (Serverless HPC over Cloud) to schedule serverless HPC workloads on the clouds. The suggested Shoc architecture integrates scheduling, scaling, and various policies to handle the whole life cycle of the jobs on different Cloud platforms such as OpenStack, Microsoft Azure, or Amazon Web Services [11].

On the other hand, due to its serverless nature, the architecture being proposed is responsible for setting up and configuring complex software like Slurm [12] or various queue management systems. Meanwhile, our approach requires only the target executable on the user's system and, optionally, a simple command-line tool for easy control that a web-based interface can easily replace.

The rest of the paper is as follows: in Section 2, the related work is presented; the background and technology stack overview are given in Section 3, while the methodology and architecture are in Section 4. Then, the proposed architecture is discussed in Section 5, while the conclusion is presented in Section 6.

2. Related work

Although using Kubernetes and containerization is not new in the HPC world, existing solutions are mostly trying to benefit from combining Kubernetes with an external resource and workload manager setup. In most cases, the approach works well, but they have significant limitations due to deployment and setup complexity.

A workflow graph approach is proposed by [13] to declare and execute complex environments onto multiple sites based on Kubernetes orchestration. The authors offer various configurations using different setups of orchestration models and underlying technologies. However, one of the main limitations of the proposed architecture is its complex deployment model and significant coupling of components, for example, in inter-container communication. Secondly, the end-user should be aware of multiple different technologies to use it.

Authors in [14] provide a comparison and detailed description of scheduling sub-systems in various technologies (Slurm, YARN, Mesos, Kubernetes) concerning HPC and Big Data requirements. The study shows several aspects of different schedulers' overall performance and resource utilization. However, there are limited measurements for Kubernetes, and it figures only in metadata comparison, particularly the primary features. Besides, the work gives a good sense of the scheduler's performances compared against each other and in different setups. Therefore, exploring Kubernetes (kube-scheduler) as a scheduler for the Shoc architecture has a vital role.

Paper [15] describes different configurations of orchestration based on Kubernetes, OpenShift, Docker Swarm and others, gives a detailed explanation of required settings for each setup and evaluates the performance. Also, the study demonstrates various key performance indicators of the arrangements based on network configurations. The conclusion insight attributes the slight overhead of Kubernetes setup due to virtualized network, which is, in fact, a trade-off between flexibility and latency. The Shoc architecture, despite this fact, chooses flexibility to enable highly scalable architecture and stay technology-agnostic.

The most recent research in this area [16] proposes a hybrid architecture of Kubernetes with Terascale Open-source Resource and QUEue Manager (TORQUE) for running and orchestration virtual HPC clusters over Kubernetes using usually the well-known Portable Batch System (PBS) notation. The proposed approach benefits from adding the TORQUE operator into the Kubernetes and utilizing both technologies, although, this adds significant complexity to the deployment. In the case of Shoc architecture, the Kubernetes-centric solution is suggested without coupling to an external resource manager.

Most studies only declare and execute complex environments, while several studies use different scheduling sub-systems in various technologies. The suggested

Shoc architecture has overcome the limitations of scheduling serverless HPC workloads on the clouds.

3. Background and technology stack overview

The Kubernetes orchestration tool is widely used to build application services for HPC and Big Data workloads. This section gives more context to the leading parallel programming, workload management, and container technologies used in the Shoc architecture.

3.1. Parallel programming paradigm

In HPC, one of the most important parallel programming paradigms is MPI [17], which is a standard library for distributed-memory parallelization. The OpenMPI [18] and MPICH [19] are the traditional implementations of the MPI standard. Though the implementations are written in C, various bindings on top of the C implementation allow access to the MPI interface from higher-level languages such as Java or Python. The proposed architecture is not limited to running MPI applications, but the MPI applications (OpenMPI) as a reference are considered.

3.2. Workload management

Simple HPC workflows demand limited resources and software configurations. Mostly, the HPC cluster is set up along with required packages, and libraries directly run the job and collect the output. However, when resources, scalability, and requirements are high, the demand for resource management, queuing, and workload management arise. The workload manager in HPC systems consists of a job scheduler and a resource manager. A resource manager allocates resources, schedules jobs, and guarantees no interference from other user processes. A job scheduler regulates the job priorities, enforces resource limits, and dispatches jobs.

Slurm is a well-known workload and resource management system consisting of a set of tools for easy scheduling and distribution of computational resources across nodes of a given HPC cluster. Slurm takes some primary responsibilities from the end-user, such as manual queuing and dedication, and assurance of computational resources. Slurm has advanced fault-tolerant scheduling capabilities and extensible architecture. It also supports containers, making the execution of HPC workloads cleaner and more manageable.

OpenPBS is another resource management solution widely used for HPC systems. One of the most critical features of OpenPBS is scalability. Besides that, OpenPBS is known for its policy-driver scheduling and plugin framework, extending functionality to meet custom requirements. TORQUE [20] is an open-source successor to OpenPBS that adds another level of fault tolerance, a more flexible scheduling interface, and scalability.

Although these workload management solutions work for the cloud-based clusters, they still require significant effort to customize and maintain in cloud environments.

3.3. Containers

Containerization creates a package consisting of the target program and its critical dependencies. Containers run on top of Operating Systems (OS) that access a shared OS kernel without needing for VMs.

Docker [21] is one of the most popular container technologies, a de facto standard for containerized applications running on the cloud. While Docker fits well for most cases due to its simplicity, popularity, and portability, there are other essential nuances to consider while choosing between containerization software for the HPC-specific needs. In particular, the requirement of privilege escalation remains the most prominent issue while using Docker for HPC.

In addition, Singularity [22] is a natural choice for HPC workflows. The Singularity may run as a regular user without privilege escalation, making it a better choice for running 3rd party applications. Singularity is compatible with Kubernetes, which means Kubernetes and Docker Containers can orchestrate singularity containers.

3.4. Container orchestration and Kubernetes

Moving toward cloud infrastructure and containerization, the need to manage thousands of containers becomes critical. It is easy to pack and run several containers on a single machine. Still, if there is a need to manage thousands of containers on hundreds of nodes at a scale, there should be an infrastructure for orchestration, resource management, and scheduling.

In this regard, Kubernetes is one of the most critical technologies. Of course, container orchestration infrastructure can be implemented not only with Kubernetes. Alternatively, Docker Swarm, Apache Mesos, and other solutions are available for achieving the same goal. In the Shoc architecture, the Kubernetes is demonstrated as a reference. However, the Shoc architecture can be easily adapted to any other container orchestration technology.

Kubernetes has rich resource management and workload scheduling functionality. Policy-based scheduling and declarative resource requirements allow building a serverless HPC solution to run HPC workloads over Kubernetes. On the other hand, Kubernetes supports several container runtime environments other than Docker. Therefore, it could be easily implemented to orchestrate Singularity-based containers, more common in the HPC world.

4. Methodology

Although a classical architecture of HPC systems based on OpenPBS, TORQUE, or Slurm as a resource manager works for many cases, there is still a big disconnect between HPC and Cloud systems. The proposed methodology of addressing the issue defines its scopes as follows:

- Reduce the complexity of an HPC deployment;
- Solve maintainability and scalability problems by reducing hardware coupling;

- Achieve a simple and seamless end-user experience working with the HPC system;
- Provide a ready-to-go solution for both public and private cloud providers,
- Enable support of auto-scaling out-of-box;
- Reduce coupling with an actual technology stack (OpenMPI, Spark, etc.);
- Enable serverless interface for HPC workload scheduling.

While targets are highlighted, it's important to define areas that are out of scope for this research:

- Integration with a third-party resource and workload managers as Slurm, OpenPBS;
- Introduce advanced scheduling capabilities that are not supported by Kubernetes but are supported by Slurm or OpenPBS;
- Provide further performance, memory, or energy optimization for MPI or other HPC workloads.

The section presents the overall methodology used to achieve the highlighted goals.

4.1. Scheduling and resource management

First, Shoc takes Slurm or OpenPBS out of the execution cycle. Hence, scheduling and resource management responsibilities lie on Kubernetes. It relies on kube-scheduler [23], which is part of the Kubernetes Control Plane. While most available scheduling features are supported by kube-scheduler, it is evident [24] that kube-scheduler wouldn't have all the Slurm scheduling capabilities. Nevertheless, most of them can be added with the Kubernetes plugin system. And, of course, using Kubernetes means also relying on it for resource management. Let us consider three main types of resources that could be allocated for the HPC workload using Kubernetes: CPU, Memory, and GPU. Another advantage of using Kubernetes as a resource manager is very granular control over-allocated resources. This enables capabilities such as topology-aware scheduling, which is critical for HPC. This way, Kubernetes can allocate resources on the same node or, if required, on multiple topologically co-located nodes or satisfy other requirements defined by a specific policy.

4.2. Containerization

The Shoc architecture can use any technology compatible with Kubernetes. However, for practical reasons, Shoc uses Docker and Singularity as the primary container runtimes for the Shoc architecture. One of the most critical advantages of Shoc architecture is that the end-user is unaware of any containers, pods, and other infrastructure-level complexities. To achieve this, Shoc provides back-end services that containerize target programs. In this case, the end-user submits an executable to the Shoc system, and a particular back-end service containerizes the given executable along with its dependencies. This level of abstraction allows containerization of any workload with the Container runtime of choice. Thus, whether it is an OpenMPI executable with dependencies or a Java-based Spark application, it gets containerized the same way. The container image of the workload is then pushed into a unique

registry for further reference. This containerization method is used as a foundation of well-known serverless systems (function-as-a-service, etc.).

4.3. Virtual clustering

In traditional HPC systems, a workload is executed over a cluster managed by the workload and resource managers. In this bare-metal provisioning, the cluster is a set of interconnected physical or virtual nodes, and its resources are shared between all workloads running on the system. The Shoc architecture offers another clustering approach, which is, in essence, another layer of virtualization to run pods (with containers inside) as a virtual cluster. So, every workload in Shoc forms a virtual cluster of pods over a Kubernetes instance. The approach, along with resource management and scheduling capabilities, allocates or requests a certain number of resources for a single workload considering policies such as topology-awareness costs or rate limits. Nodes in the pods are allocated across the nodes in a Kubernetes cluster. Moreover, this allows workloads to run not over a single Kubernetes cluster but multiple. To achieve this, the Shoc back-end service maintains a set of Kubernetes cluster references and submitting another workload can be further load-balanced or allocated regarding regional considerations, general availability, redundancy, and underlying container runtime.

4.4. Auto-scaling

So far, the paper describes a methodology of running virtu-ally clustered workload over a Kubernetes instance. Then, as part of the methodology, it states that the Shoc system is backed by more than one Kubernetes instance, enabling manual scaling. There are, however, other aspects of scaling that Shoc architecture enables. Of course, every underlying Kubernetes instance can be scaled manually by adding new nodes to the kube cluster. Still, there is another feature for Kubernetes which can play a critical role in such an HPC ecosystem. The kube-autoscaler feature allows the Kubernetes cluster to instantiate and join a node on demand, enabling Shoc to support massive scale infrastructures. This way, if Shoc is given several Kubernetes cluster references managed by various Cloud providers (public or private), it will be ready to scale up or down based on the actual resource usage. This makes Shoc a serverless system, as underlying resources are allocated and de-allocated without the involvement of the Shoc system or a human operation.

5. Shoc architecture

The Shoc architecture comprises micro-services, which deploy within the Kubernetes cluster, hosted by any public or private cloud environment. The same Kubernetes cluster can execute requested HPC workloads. However, any number of clusters will then be connected to the system for a broader resource spectrum. Figure 1 shows the essential components of the architecture, including the end-user side, container registry, container engine, builder, executor, and Kubernetes auto-scaling.

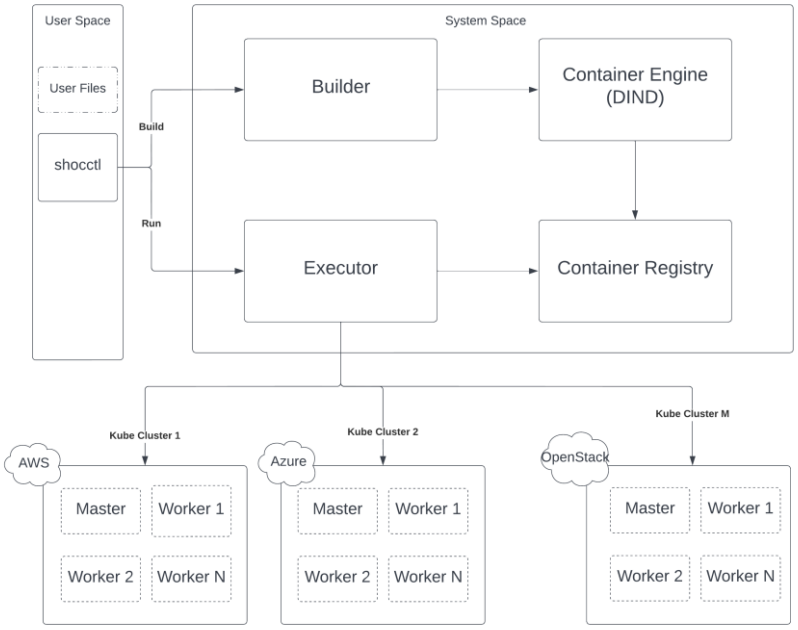


Fig. 1. The overview of Shoc architecture.

5.1. The end user's side

The end-user is unaware of containerization and container orchestration technologies, even though Shoc relies on technologies like Docker and Kubernetes. Consequently, there is no need to install any piece of software on the end-user side, besides a command-line application capable of controlling the whole workflow on the local machine shocctl. As an alternative solution for the end-user, a web-based client application can be a great replacement to avoid installing software on a machine. The command-line or web-based interface on the end-users side only communicates with the back-end system using REST-full API calls, which means the process can be automated further by introducing more clients such as shell scripts.

The end-user application collects the necessary HPC executable, dependencies, and input files and then sends those over to the back-end for further processing.

5.2. The container registry and the engine

As the end-user only deals with HPC executable files and relies on containerization to place the workload in the system, the solution needs a centralized registry to store all the containers built for every revision of the workloads. This registry component is naturally designed as a micro-service wrapping a private Docker Registry to store all the images built. Docker Registry exposes a set of unique REST-full endpoints so that the Container Engine and the Kubernetes can push and pull images over HTTP.

The Shoc system turns raw executable files and libraries into containers. For this purpose, a container engine builds images based on Dockerfile or similar instruction files, then pushes the created image into the container registry. Choosing

Docker as a primary container engine, the Docker-in-Docker will be used as an approach to building out the micro-service that exposes required endpoints for container manipulations (building images, placing them into the registry, etc.).

5.3. The builder and executor

Another central service in the Shoc system is the builder generating a Dockerfile or alternative recipe for the given HPC package. This service is the leading virtualization point responsible for analysing the HPC job's nature and generating the required instructions to build the final container image. Building a container image for a compute-intensive MPI-based HPC workload significantly differs from building an image for a data-intensive Spark-based application. Thus, this tier will unify all the supported HPC workload types and make them available for execution.

The workload is available for execution after the builder finishes building a ready-to-execute container image. The next micro-service handles the process of actual execution (workload placement) in the pipeline called an executor. This service is responsible for placing a user cluster for running the actual job. Running a workload, the system needs to account for several important aspects: required resources, priorities, virtual node topology, and output collection methods. Mentioned aspects are given as particular arguments to the API exposed by the service. Every workload is submitted as a virtual cluster to one of the well-known Kubernetes instances. The virtual cluster is modelled then as a StatefulSet or Deployment relying on workload image. All the resource and other requirements (including topological and other policies) are evaluated against referenced Kubernetes instances and then submitted to the best fitting instance. Of course, this may lead to queuing jobs inside Kubernetes as requested resources may not be available at the time. In this case, the system relies on kube-scheduler for prioritization and Deployment/StatefulSet placement.

5.4. The Kubernetes auto-scaling

While Shoc may refer to several Kubernetes instances for scaling purposes, finite resources can still be a limitation. For this case, it relies on an out-of-box solution for Kubernetes auto-scaling. With Kubernetes Cluster Auto-Scaler integrated into the instance, the cluster may utilize the cloud provider's API to dedicate more resources. Most well-known global cloud providers such as Amazon AWS, Azure, Google Cloud Engine, Google Kubernetes Engine, OpenStack, etc. are supported out-of-box.

5.5. The communication models

As the main components are defined, it is essential to describe the communication model. Naturally, the software on the end-user's side (shocctl, web-based interface) communicates with the Shoc system over a unique API exposed by Shoc. Functionality that should be exposed to the end-user in Shoc is exposed with a set of REST-full APIs. Internal communication between different components such as builder and container engine or executor and Kubernetes instance is done over APIs exposed by third-party software (Docker Engine, Kubernetes).

6. Conclusion

The article presents the architecture of the Shoc system. Shoc architecture aims to advance seamless cloud infrastructure usage for running HPC workloads by benefiting from modern cloud technologies. It adds serverless experience to the end-user and takes out the complexity of deploying HPC infrastructures. The proposed methodology expands existing high-performance computing technologies with containerization enabling seamless scaling, deployment, and clustering capabilities.

The further implementation and the experiments will rely on the computational and storage resources of the Armenian research cloud infrastructure [25]. The multi-agent algorithms and systems will be developed in Shoc to process the images received from the self-organized swarm of unmanned aerial vehicles.

***Acknowledgement*:** This work is partially supported by the EC Horizon2020NI4OS-Europe (National Initiatives for Open Science in Europe) Project (Nr. 857645) and the "Self-organized Swarm of UAVs Smart Cloud Platform Equipped with Multi-Agent Algorithms and Systems" Project (No 21AG-1B052) supported by the Armenian State Committee of Science.

References

1. Giles, M. B., I. Reguly. Trends in High-Performance Computing for Engineering Calculations. – Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences (The Royal Society Publishing), Vol. **372**, 2014, 20130319.
2. Dagum, L., R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. – IEEE Computational Science and Engineering (IEEE), Vol. **5**, 1998, pp. 46-55.
3. Walker, D. W., J. J. Dongarra. MPI: A Standard Message Passing Interface. – Supercomputer (ASFRA BV), Vol. **12**, 1996, pp. 56-68.
4. Buyya, R., C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. – Future Generation Computer Systems (Elsevier), Vol. **25**, 2009, pp. 599-616.
5. Giunta, G., R. Montella, G. Agrillo, G. Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. – In: Proc. of European Conference on Parallel Processing, 2010, pp. 379-391.
6. Abid, M. R. HPC (High-Performance the Computing) for Big Data on Cloud: Opportunities and Challenges. – International Journal of Computer Theory and Engineering (IACSIT Press), Vol. **8**, 2016, 423.
7. Zhang, J., X. Lu, D. K. Panda. Is Singularity-Based Container Technology Ready for Running MPI Applications on HPC Clouds? – In: Proc. of 10th International Conference on Utility and Cloud Computing, 2017, pp. 151-160.
8. Rosado, T., J. Bernardino. An Overview of Openstack Architecture– In: Proc. of 18th International Database Engineering & Applications Symposium, USA, New York, Association for Computing Machinery, 2014, pp. 366-367.
9. Abdelbaky, M., J. Diaz-Montes, M. Parashar, M. Unuvar, M. Steinder. Docker Containers across Multiple Clouds and Data Centers. – In: Proc. of IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC'15), 2015, pp. 368-371.
10. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. – IEEE Cloud Computing, Vol. **1**, 2014, pp. 81-84.
11. Serrano, N., G. Gallardo, J. Hernantes. Infrastructure as a Service and Cloud Technologies. – IEEE Software, Vol. **32**, 2015, pp. 30-36.
12. Yoo, A. B., M. A. Jette, M. Grondona. SLURM: Simple Linux Utility for Resource Management. – In: Job Scheduling Strategies for Parallel Processing. Berlin, Heidelberg, Springer, 2003, pp. 44-60.

13. C o l o n n e l l i, I., B. C a n t a l u p o, I. M e r e l l i, M. A l d i n u c c i. StreamFlow: Cross-Breeding Cloud with HPC. – IEEE Transactions on Emerging Topics in Computing, 2020, pp. 1-1.
14. R e u t h e r, A., et al. Scalable System Scheduling for HPC and Big Data. – Journal of Parallel and Distributed Computing (Elsevier BV), Vol. **111**, January 2018, pp. 76-92.
15. B e l t r e, A. M., P. S a h a, M. G o v i n d a r a j u, A. Y o u n g e, R. E. G r a n t. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. – IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC'19), IEEE, 2019.
16. Z h o u, N., et al. Container Orchestration on HPC Systems through Kubernetes. – Journal of Cloud Computing (Springer Science and Business Media LLC), Vol. **10**, February 2021.
17. G r o p p, W., E. L u s k, N. D o s s, A. S k j e l l u m. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. – Parallel Computing (Elsevier BV), Vol. **22**, September 1996, pp. 789-828.
18. G a b r i e l, E., et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. – In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg, Springer, 2004, pp. 97-104.
19. G r o p p, W., E. L u s k. User's Guide for MPICH, a Portable Implementation of MPI. User's Guide for MPICH, a Portable Implementation of MPI. Citeseer, 1996.
20. S t a p l e s, G. Torque Resource Manager. – In: Proc. of ACM/IEEE Conference on Supercomputing, 2006.
21. M e r k e l, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. – Linux J. (Belltown Media), 2014, March 2014.
22. K u r t z e r, G. M., V. S o c h a t, M. W. B a u e r. Singularity: Scientific Containers for Mobility of Compute. – In: Attila Gursoy, Ed. PLOS ONE (Public Library of Science (PLoS)). Vol. **12**. May 2017, e0177459.
23. M a r t i n, P. Control Plane Components. – Kubernetes: Preparing for the CKA and CKAD Certifications, Apress, Berkeley, CA, 2021, pp. 11-13.
24. Slurm vs. LSF vs. Kubernetes Scheduler: Which is Right for You? (Online). **https://www.run.ai/guides/slurm/slurm-vs-lsf-vs-kubernetes-scheduler-which-is-right-for-you**
25. A s t s a t r y a n, H., et al. Strengthening Compute and Data Intensive Capacities of Armenia. – In: Proc. of 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER), 2015, pp. 28-33.

Received: 14.02.2022; Second Version: 12.06.2022; Accepted: 28.06.2022