Optimizing Build Time of Shoc Workloads

Davit Petrosyan
Institute for Informatics and Automation Problems of NAS RA
Yerevan, Armenia
e-mail: davit.petrosyan@iiap.sci.am

Harutyun Mkrtchyan
Yerevan State University
Yerevan, Armenia
e-mail: harutyun.mkrtchyan2@edu.ysu.am

Hrachya Astsatryan
Institute for Informatics and Automation Problems of NAS RA
Yerevan, Armenia
e-mail: hrach@sci.am

***Abstract*—Running HPC workloads can be challenging due to infrastructure complexity, the learning curve and the specifics of various workload management and scheduling solutions. The paper aims to describe the recent advancements in the Shoc [1] architecture designed for submitting serverless HPC workloads on cloud infrastructure. In particular, the paper discusses the set of architectural advancements in the performance of the build process.**

***Keywords*—HPC, containerization, cloud, Kubernetes.**

I. Introduction

The increasing demand for high-performance computing (HPC) workloads in cloud environments has presented opportunities and challenges in recent years. While cloud computing offers scalable and flexible resources for executing compute-intensive tasks, the complexity of managing cloud infrastructures and orchestrating diverse HPC workloads remains a significant hurdle.

As discussed in an early work [1], the proposed architecture models a serverless approach to submitting high-performance workloads to cloud environments. The main aim of the design is to reduce the complexity of working with cloud environments so that the end user running a cloud job is aware of the algorithm. In contrast, the complexity of infrastructure setup, job scheduling, and resource management is something the Shoc infrastructure should take care of.

The diversity of technologies, runtimes, and libraries employed in creating HPC jobs for various use cases further compounds the challenges of deploying tasks in the same cloud environment. To surmount this obstacle, our architecture leverages container technologies for workload encapsulation and employs container orchestration technologies to manage workloads and resources in the cloud infrastructure effectively.

The article presents an overview of the pipeline employed by our proposed architecture to facilitate the seamless execution of HPC workloads in the cloud. The critical steps in the pipeline are as follows:

- The process begins by preparing the source code of the HPC task, which is possible to implement in various programming languages such as C/C++, MPI [2], Python, and others;
- We containerize the source code by building a Singularity [3] or Docker [4] image for the solution to encapsulate the HPC solution effectively. Containerization ensures the consistency of the execution environment, irrespective of the underlying infrastructure;
- The containerized solution's image is then utilized to create a workload that is submitted to the container orchestration system, such as Kubernetes [5]. The orchestration system efficiently manages the deployment and scaling of containers across the cloud infrastructure;
- The orchestration system ensures that all the prerequisites for the HPC job, including CPU, memory, and other conditions, are met before initiating the execution of the workload;
- Upon successful execution, the results of the HPC job are collected, completing the pipeline.

II. Architecture Overview

The Shoc architecture comprises several integral components that work synergistically to facilitate seamless deployment and execution of HPC workloads in cloud environments. Each part plays a distinct role in achieving the architecture's overarching goal of simplifying cloud-based HPC for end users. The key components are as follows:

- The shoccli is the user interface, enabling users to interact with the Shoc infrastructure effortlessly. Through a straightforward command-line interface, users can submit their HPC tasks, manage workloads, and monitor job execution, abstracting the complexities of the underlying infrastructure;
- The back-end (builder) service is responsible for containerizing the HPC workload. By encapsulating the solution's source code into container images, this service ensures a consistent and reproducible execution environment, irrespective of the cloud infrastructure's specifics;
- The architecture utilizes a container engine (such as docker-in-docker), which efficiently manages the execution and resource allocation for individual containers;
- Shoc incorporates a built-in image registry to store the container images created by the builder service. This centralized repository facilitates the retrieval and sharing of pre-built images, streamlining the execution of subsequent tasks;
- The executor service takes charge of executing the workloads prepared by the builder. It is responsible for submitting jobs to the cloud environment, managing task execution, and monitoring job progress;

https://doi.org/10.51408/csit2023_20

- As a middleware between the front-end (shoccli) and the back-end services, the reverse proxy enables seamless communication and data transfer between the user interface and the underlying components.

The infrastructure also provides a framework for adding handlers for various workloads. The current architecture already covers several use cases:

A. Single-Node Workloads

The Shoc infrastructure offers a seamless and straightforward approach to running serverless HPC workloads for algorithms written in various supported languages, including Python, Node.js, .NET, Java, C/C++, and more. End users can now submit their source code, and the Shoc back-end takes care of the rest.

Upon receiving the source code, the Shoc back-end employs a specialized handler tailored to the corresponding technology to build a comprehensive workload image. This image includes all the necessary files, dependencies, libraries, and input sources essential for the workload's successful execution. The back end generates a recipe specific to the workload type. It utilizes a hosted container engine to create an OCI-compatible image, which is then stored in the built-in or external image registry.

Once the image is successfully built, the next step involves deploying the newly created image to the underlying orchestration infrastructure. The Shoc system's executor module handles this task using another specialized handler designed to deploy the job effectively. For instance, if the Shoc system has one or more Kubernetes clusters registered, the handler will select an appropriate cluster, generate a native Kubernetes Batch Job object, and submit it to the cluster for execution. Subsequently, the system attaches a listener to specific pods or containers associated with the job, ensuring seamless output collection and its prompt return to the end user.

As a result, end users can now submit their algorithms written in their preferred language to the Shoc infrastructure with a single command and swiftly obtain the desired results. The Shoc infrastructure and the underlying orchestration technology, such as Kubernetes or Mesos [6], expertly manage all the intricacies of scheduling and resource management.

B. Multi-Node Workloads

While existing cloud-native technologies like AWS Lambda and Azure Function facilitate the execution of single-node serverless jobs, they do not support the submission of multi-node workloads, such as those relying on MPI-based algorithms or Spark jobs [7].

Shoc infrastructure offers a powerful solution that transforms these complex multi-node jobs to address this limitation, enabling them to run over virtualized clusters. With this approach, Shoc provides a built-in capability to submit HPC workloads with specific resource requirements, while maintaining transparency for the end user.

Shoc achieves this by creating a set of prerequisite objects in the underlying orchestration system, effectively establishing a virtual cluster to accommodate the multi-node workload. For instance, a Kubernetes cluster registered within the Shoc system will create a prerequisite StatefulSet with defined resource requirements such as CPU, RAM, GPU, and more. Once the StatefulSet is successfully deployed to the cluster, the main job can utilize pods as a set of virtual cluster nodes, effectively executing the multi-node workload.

By seamlessly abstracting the complexities of virtual cluster setup and resource allocation, Shoc empowers users to effortlessly submit multi-node HPC workloads without requiring intricate manual configurations. Researchers and developers can now focus entirely on their algorithm's design and functionality, secure in the knowledge that Shoc's intelligent orchestration will handle the efficient execution of their multi-node tasks.

This innovative capability extends the possibilities of cloud-based HPC, facilitating the deployment of sophisticated algorithms and data-intensive processing, and opens new avenues for scientific exploration and discovery. The Shoc infrastructure's adaptability and versatility make it a valuable tool for researchers across diverse domains, propelling advancements in cloud-based HPC.

III. Problem Statement

As per Shoc's design, the end user should have only a small Shoc CLI tool installed on its environment, and the rest of the pipeline, from compiling the solution to the execution, is the primary concern of the Shoc back-end. The drawback of such a degree of flexibility is that the source code and any other assets (such as input files, etc.), should be sent to the server before every execution. Moreover, once the server gets the new set of source and data files, it needs to build a container image that will have installed all the necessary libraries, tools, compilers, etc.

This paper aims to minimize the need for solution rebuilds when not required. For example, if the end user wants to rerun the same solution with the same or other run-time parameters, the existing image could be reused. On the other hand, if any modifications in the solution don't affect the final executable solution, the existing image will be reused.

IV. Solution

A possible way to address the mentioned problem is to keep track of the modifications on the system at multiple stages.

Assuming the user wants to submit the solution directory *D* that contains source files s_1, s_2, ..., s_n and data files d_1, d_2, ..., d_n. Depending on its underlying technology, every type of workload will require a different subset of the solution files. For example, if the solution is given with binary files and does not need the build process to be involved, the files containing source code are not required for the build process. And the opposite, if the solution type requires only source code to be available, the compiled binaries, libraries, and other dependencies are unnecessary, as the build pipeline will take care of those on the back-end side.

For example, while building a Python package, sending the solution's dependencies to the server is unnecessary, as the pipeline can download the dependencies with only the requirements.txt file.

A. Ignoring unnecessary files

One advancement towards the mentioned optimization is ignoring the files that do not affect the build pipeline. To achieve that, Shoc allows the end users to create a special *.shocignore* file and list specific rules for ignoring unnecessary files.

Every rule in the file is a text line representing one of the following:

- the path of the file or a directory,
- the pattern of file or a directory.

By this, while the CLI tool creates a bundle for sending to the server, it will not include files or directories matching any rule in the .shocignore file. Hence, the time to archive the bundle, network traffic, and overall transfer speed will be significantly reduced. It's pretty common to exclude directories such as *.git, node_modules, target, bin, obj, etc.* as they don't affect the build process while having a large size.

B. Hashing bundle listing

We could reduce the bundle size and speed up the upload process by ignoring unnecessary files. The upload still happens, so the build process is triggered even if no file was modified.

There are several approaches to address the issue, however, it is preferable that the optimization be transparent to the end user. Therefore, the next optimization was applied to the Shoc pipeline to prevent bundle upload if there is no need.

To make sure the optimization is valid we make the following assumptions:

- if input files are not modified since the last build, the resulting image should be the same,
- the result of the build process does not depend on anything other than input files,
- in case of any non-reproducible build stages, the rebuild process should be forced.

Based on the mentioned assumptions the CLI application working on the end user's side will make a pre-flight check as follows:

- list all the files required for the bundle (according to .shocignore file),
- build a special temporary file containing the list of the source files along with their modification timestamps,
- computes the hash of the listing file and uses it as a lookup key,
- in case there is a known bundle with the given listing hash, assume the image is already built.

This way, the system will not upload the bundle to the server if it could find an image built with the same set of files.

Obviously, the method has some downsides as the timestamp change does not always reflect the change in the content.

C. Hashing the bundle

Applying the mentioned optimizations, the system reduces the need to send bundles over and over again, if there is no need. However, in some instances, there is a chance that some of the files will have a modified timestamp, however, the content will remain the same.

To address the problem, the system will apply the next level of pre-flight check. This time, the system will allow bundling to be generated on the end user's machine and after the bundle file is created, the system will calculate the hash of the overall bundle again.

The generated hash code will be used as a lookup key for the bundle in the system. So, if there is an image in the system having the same bundle hash, the system will not send the bundle to the server assuming that the resulting image already exists and does not need to be updated.

In this way, the system will optimize both upload and build time (compilation, container image creation, etc.).

D. Container image caching

A combination of all the above optimizations will guarantee that replaying the same workload with the same or another set of execution parameters will not trigger the whole build pipeline, which can take a long time to complete due to the following reasons:

- the bundle is archived on the end user's machine,
- the bundle is uploaded to the server,
- the bundle is used to build a container image,
- image build process can take a long time to load its parent image and install dependencies.

To reduce the time of loading potentially large parent images in the pipeline build stage, the system will utilize the filesystem layering approach used by modern container runtimes such as Docker and Singularity.

To enable layering behaviour, a special service in the Shoc ecosystem called *dind (docker-in-docker)* will have permanent storage attached to it, so that, if the parent image of the specified version was loaded once, it will be automatically mounted on the next usage.

V. Conclusion

Adding a variety of use cases to the Shoc infrastructure increases its overall architectural complexity, leading to sub-optimal performance in different scenarios.

The article presents recent advancements and optimizations made to the Shoc system allowing to significantly improve the speed of build process in the pipeline. The proposed methodology uses several layers of heuristics as well as uses built-in optimizations of the referenced technologies to increase the performance of the build process and increase overall convenience for the end user.

The further implementation will rely on the mentioned techniques as well as expand area of performance and stability improvements.

ACKNOWLEDGMENT

This work is partially supported by the EC Horizon2020 NI4OS-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 (Nr. 21AG-1B052) supported by the Armenian State Committee of Science.

REFERENCES

[1] D. Petrosyan and H. Astsatryan, "Serverless high-performance computing over cloud," *Cybernetics and Information Technologies*, vol. 22, no. 3, pp. 82–92, 2022. [Online]. Available: https://doi.org/10.2478/cait-2022-0029

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996. [Online]. Available: https://doi.org/10.1016/0167-8191(96)00024-5

[3] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, p. e0177459, May 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[4] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[5] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[6] M. Frampton, "Apache mesos," in *Complete Guide to Open Source Big Data Stack*. Apress, 2018, pp. 97–137. [Online]. Available: https://doi.org/10.1007/978-1-4842-2149-5_4

[7] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," 2017.