Performance Characterization of Containerized HPC Workloads

Maximilian Höb[a,*]

[a]*MNM-Team, Leibniz Supercomputing Centre and Ludwig-Maximilians-Universität München, Germany*

E-mail: hoeb@mnm-team.org

In this paper, we address the complex challenges that arise within existing high performance computing (HPC) frameworks as we approach the exascale era. On one side, highly optimized, heterogeneous hardware systems coexist with HPC-unexperienced scientists with increasing demand for compute and data capacity. We propose containerization as a key concept to shift the focus back to the actual domain science, enabling an efficient usage of the compute systems and removing incompatible dependencies, unsupported subprograms or compilation challenges. To this end, we provide a methodology to determine, analyze and evaluate characteristic parameters of containerized HPC applications to fingerprint the overall performance of arbitrary containerized applications. The methodology comprises the performance parameter definition and selection, a measurement method to minimize overhead, and a fingerprinting algorithm to enable characteristics comparison and mapping between application and target system. We apply the methodology to benchmark applications to demonstrate its capability.

International Symposium on Grids & Clouds (ISGC) 2023 in conjunction with HEPiX Spring 2023 Workshop, ISGC&HEPiX2023
19 - 31 March 2023
Academia Sinica Taipei, Taiwan

*Speaker

1. Introduction

Approaching the exascale era, complex challenges arise within the existing high performance computing (HPC) frameworks. Highly optimized, heterogenous hardware systems on the one side, and more often HPC-unexperienced scientists with a continuously increasing demand for compute and data capacity on the other side. Bringing both together would enable a broad range of scientific domains to enhance their models, simulations and findings while efficiently using the existing and future compute capabilities. Those systems will continue to develop a more and more heterogeneous landscape of compute clusters, varying classical computation and accelerator cores, interconnects or memory and storage protocols and types. Adapting user applications to those changing characteristics is laborious and prevents enhancing the core functions of the applications while focusing on deployment and runtime issues. Consequently, containerization is one key concept to shift the focus back to the actual domain science, removing incompatible dependencies, unsupported subprograms or compilation challenges. Additionally, an optimized efficiency of the compute systems' usage is reachable, if system owners would be aware of the actual requirements of the containerized applications.

This work in progress-paper bases on a methodology to determine, analyze and evaluate characteristic parameters of containerized HPC applications to fingerprint the overall performance of arbitrary containerized applications. The methodology comprises the performance parameter definition and selection, a measurement method to minimize overhead, and a fingerprinting algorithm to enable characteristics comparison and mapping between application and target system. By applying the methodology to benchmark applications we aim to demonstrate its capability to reproduce expected performance behavior. Future work will build prediction models of the application's resource usage within a certain trash-hold. Thereby, we will enable a twofold enhancement of today's HPC workflows, an increase of the system's usage efficiency and a runtime optimization of the application's container. The system's usage efficiency will be enabled by container selection and placement optimizations based on the container fingerprint, while the runtime will profit from a streamlined, target-cluster-oriented allocation and deployment to optimize time-to-solution.

The most prospective technology to overcome endless adaptions of the application's program code is containerization, which offers portability among heterogeneous clusters and unprecedented adaptability to target cluster specifications. Containers like Singularity[1], Podman[2] or Docker[3] are well known for cloud usage and micro-service environments. During the last years containers like Apptainer[4] or Charliecloud[5] became also widespread in certain HPC domains, since their capabilities to include high data throughput, intra- and inter-node communication, as well as the overall scalability increased enormously.

We base our approach on the EASEY (Enable exASclae for EverYone in [1]) framework, which can automatically deploy optimized container computations with negligible overhead. Todays containers are natively not able to automatically use all given hardware at best, since the encapsulated application varies on computing, memory or communication demands. An added abstraction layer,

[1]`https://sylabs.io`
[2]`https://podman.io`
[3]`https://docker.com`
[4]`https://apptainer.org`
[5]`https://hpc.github.io/charliecloud/`

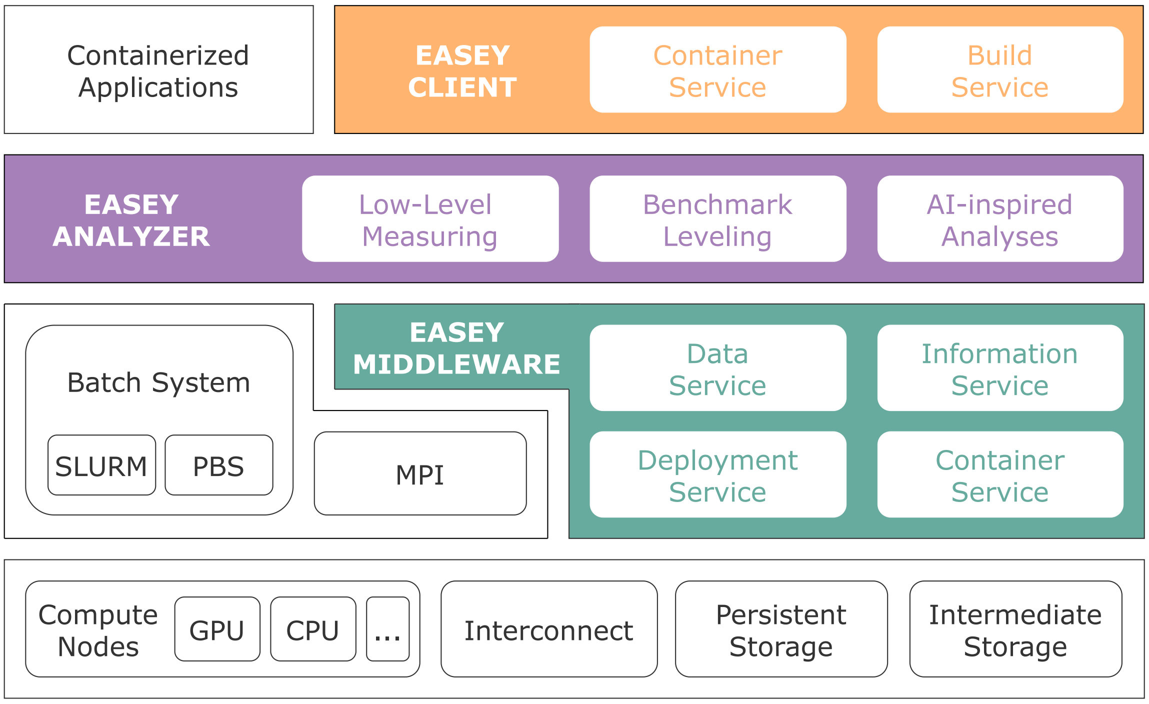

Figure 1: Updated EASEY layered architecture

although enabling many programming models and languages to be executed on very different hardware, is not able to make use of all provided hardware features. An enhanced EASEY framework will support distinct optimization tunings by determining the containers runtime characteristics.

The remainder of the paper is divided into three sections. Section 2 gives an overview on the EASEY framework. The enhanced EASEY Analyzer is presented in Section 3. The fingerprinting algorithm and the characterization is detailed in Section 4, which will also present the initial measurements and evaluation. Section 5 concludes the paper.

2. EASEY Background

In [1] we introduced a deployment framework, which enables scientist to easily convert a docker container to a charliecloud container while including specific target system dependencies and libraries.

This is especially important since more and more HPC-unexperienced scientist want and need to use supercomputing facilities, because their own smaller system are no longer capable of their own computation. With them reaching out to large and vey large clusters different hurdles need to be overcome. The main goal of the before mentioned paper and its framework was to enable scientists to focus on their actual scientific work within their domain and reduce deployment overhead where possible. Since it is mandatory for any application on such high performance systems to adapt at least in some parts to the target system, this work was earlier done manually between application support experts and the developers. With EASEY we are able to add several optimizations while transferring the initial docker file to a charliecloud container.

The initial architecture of the EASEY framework as presented in [1] was updated with a new layer, the *EASEY ANALYZER* and is displayed in Figure 1 on page 3. Together with the existing building bricks it was integrated in the layered HPC architecture. On the upper layer, *Applications and Users*, the so called *EASEY CLIENT* is the starting point for any process and responsible for the transformation of the original dockerfile-based container. Within this transformation process the user or application owner can configure data mount points and detail the inclusion of the target system's MPI library for example. In this mandatory, json-based configuration file, it is also possible and necessary to detail deployment several specifications:

- **Job Specification**: job meta-data like name or id
- **Data Specification**: source, protocol, authentication and mount-point
- **Deployment Specification**: number of nodes, ram, cores-per-task, asks-per-node and clock-time
- **Execution Specification**: serial or mpi-based execution commands inside the container

With all these information a functional Charliecloud container can be build and additionally a slurm- or pbs-based deployment script is created. Before the actual data processing and deployment is started we introduce a new layer, the *EASEY Analyzer*. The tasks of the three submodules *BPF Analyzer*, *Benchmark Leveling* and *EASEY AI* focus on the analysis and characterization of the before created container and are detailed in Section 3.

The *EASEY Middleware* on the *local resource management layer*, as detailed in [1], is responsibly for the preparation of the execution environment, the data stage-in or -out, and the actual deployment through the local scheduler. It also offers monitoring information and status updates of the execution. Any optimization of the bare metal, the *hardware layer* of the cluster underneath is subject to future work as detailed in [1].

For more information on the details of the configuration of the EASEY workflow, we recommend to study our original paper in [1]. In there we also presented related work to this approach.

3. EASEY Analyzer

Introducing logic modules into the EASEY framework accelerated the initial deployment tool to complex framework capable of black-box analyzation and decision making. The new analyzer layer displayed in Figure 1 on page 3 consists of three main elements:

- Low-Level Measuring
- Benchmark Leveling
- AI-inspired Analyses

Those modules are able to automatically analyze the characteristics of a container and map it to the most likely target system. They can also be integrated in a logical workflow presented in Figure 2 on page 5. Included between the EASEY client and middleware, it offers within one iteration

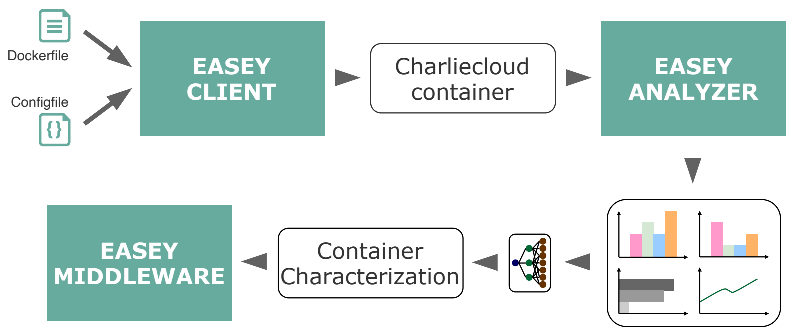

Figure 2: Schematic EASEY workflow of a job submission on a target system

the possibility for such a characterization and therefore, a direct recommendation on which kind of system this container should be deployed.

Additionally, this layer offers a module consisting of several benchmarks to determine the classification of the target system, which result will be used within the middleware to map container and system.

3.1 Low-Level Measuring

Profiling the runtime behavior of an unknown application is still ongoing research. Adding to this challenge a container framework makes it even more complex. The Berkeley Paket Filter (BPF) or the extended Berkeley Paket Filter (eBPF) offer a low-level tap point to extract necessary measurements and data while executing the application [2].

BPF was developed for UNIX to improve network monitoring application performance. As described in [3] BPF has been part of the Linux kernel since its original implementation, and has been continuously improved, leading to eBPF with a network specific architecture. eBPF is designed to be a general-purpose filtering system and can be used for applications such as packet filtering, traffic control/shaping, and tracing. The eBPF instructions are mapped to real assembly instructions of the underlying hardware architecture, and eBPF programs are verified to ensure they cannot compromise or block the kernel. eBPF programs are restricted to reading from and writing to key-value stores called maps, which are areas in memory set up by user space helpers before the eBPF program is loaded into the kernel. Data can then be accessed securely from user and eBPF kernel space.

Within this paper we use eBPF to measure the compute and memory usage of the benchmarks to gather data for the evaluation.

3.2 Benchmark Leveling

Determining the specifications of a target system builds the basis to perform a successful map of characterized containers to different clusters. They are in most cases optimized systems for specific computations and loads. Also one general purpose system might differ from another system, which should be determined in its characteristics. Therefore, we consider it mandatory to also analyze each target system included into the EASEY framework.

In this first stage of this analysis framework we focused only on CPU- and memory-heavy micro benchmarks. On the target systems, we executed two benchmarks based on the *seven dwarfs* presented in [4].

3.3 AI-inspired Analyses

Based on the measurements and data gathered from the *Low-Level Measuring* also predictions based on machine learning modules can be includes. At this stage we did not include any prediction of the performance. This artificial intelligence module is subject to future work and can be exchanged by any logical module which is able to analyze the input data to extrapolate its charateristics.

4. Fingerprinting

The overall performance of a containerized applications is difficult to classify, since many applications very in the usage of the available hardware during its execution. Many different phases can be observed while analyzing a long running application. Our approach for such a characterization is a fingerprint of the container execution, determined in four dimensions.

4.1 Characterization

We define four main dimension we consider for the performance characterization:

1. Compute
2. Memory
3. Network
4. I/O

Compute characterizes the usage of the available CPUs, which can vary extremely during the execution of HPC applications.

Memory investigates the relative usage of the available memory on the system. Due to the host system's memory usage, a trash hold has to be defined, which enables to focus only on the application's memory consumption.

Network is defined as the communication between nodes. Although HPC application are also parallelized within one node, this dimension focuses only on the actual combination over the interconnect.

I/O might also be considered as a combination of network and memory, however, we determined many applications are directly affected by I/O and added it as a fourth dimension. In future assessment, we will consider whether this dimension is mandatory or can also be expressed as a conditional dimension of the others.

To enable the classification to be comparable, we have defined only four possible levels for each dimension. Each level is represented by an integer, since it is mathematically evaluated at a later stage.

Table 1: Dimensional levels of the Fingerprinting Evaluation

Level	Compute	Memory	Network	I/O
0	~~none~~	~~none~~	none	~~none~~
1	scarce	~~scarce~~	scarce	scarce
2	moderate	moderate	moderate	moderate
3	extensive	extensive	extensive	extensive

In Table 1 on page 7 the four dimensions of the evaluation are displayed for each dimension. As it can be seen, for *Compute* and *I/O* the level *none* is crossed out, since we do not consider applications having no compute or no I/O part. As mentioned in the section before, *memory* needs to be determined relatively to the memory usage of the host system. Therefore we consider here only two dimensions, implicating a memory usage above a certain trash hold. Experiments need to prove that this approach is valid.

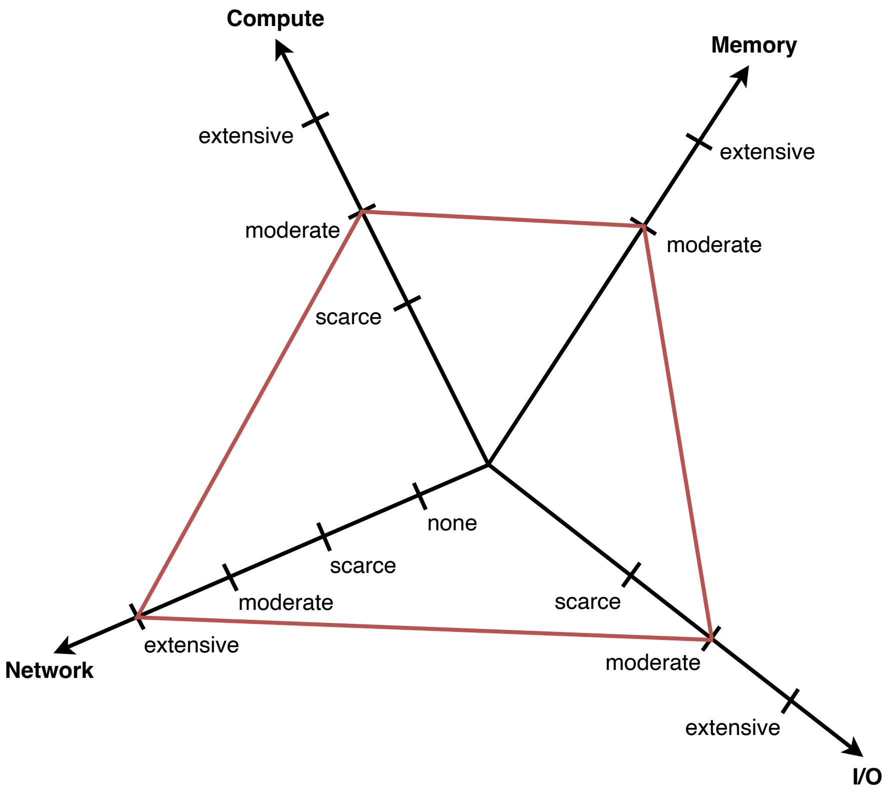

Figure 3: Dimensional Star of the Fingerprinting Evaluation's Dimensions

Figure 3 on page 7 shows a characterization of an application having a moderare compute level (2), a moderate memory level (2), an extensive network level (3) and a moderate I/O level (2). The resulting vector of this characterization can be specified as:

$$\begin{pmatrix} compute \\ memory \\ network \\ I/O \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 3 \\ 2 \end{pmatrix}$$

This vector representation can be extended with more classifications to a matrix representation of the fingerprint. However, it is work in progress to determine efficient but still significant time steps, in which those single measurements are combined. It is also subject to future work to compare mean values to time progressive interval mean values as well as the length of the corresponding intervals.

4.2 Initial Measurements

We base our measurements on compute and memory intensive benchmarks based on the seven dwarfs. Those benchmarks have been execute in several configurations and the CPU and memory usage have been recorded by kernel traces using eBPF. In Figure 4 on page 8 we show the measurements and the according classifications from a combined execution of the same benchmark using a different parallelism degree.

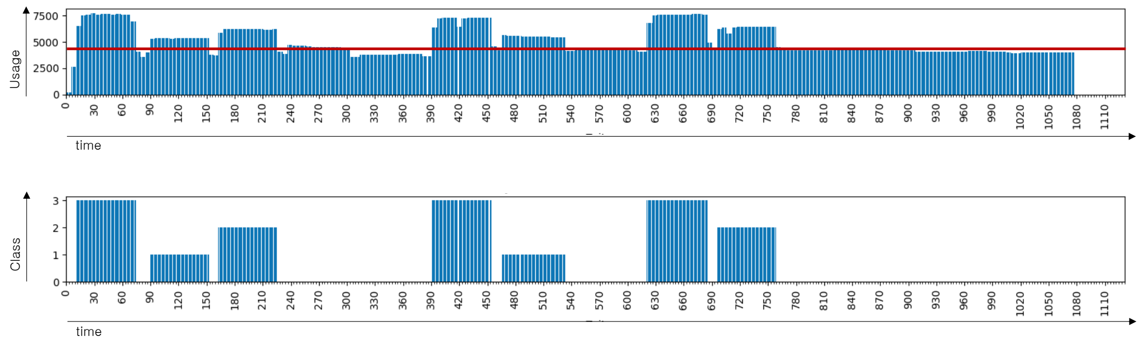

Figure 4: Measurement and Classification of Memory Usage over time

As it can be seen, the memory usage does not fall below a trash hold indicated by the red line in the upper part of the figure. Removing this memory usage of the host system, we characterized the application run for each individual time step. As mentioned in the section before, the length and potential summation of time steps need to be investigated as future work.

The correlation between dimensions is also subject to investigation. Figure 5 on page 9 shows the CPU usage classification and Figure 6 on page 9 shows the memory usage classification of the same execution, measured in parallel. It can be seen, that the memory classification correlates with an average of the CPU classification. This execution consists of eleven phases with the same benchmark in different configurations and with different parallelism degrees.

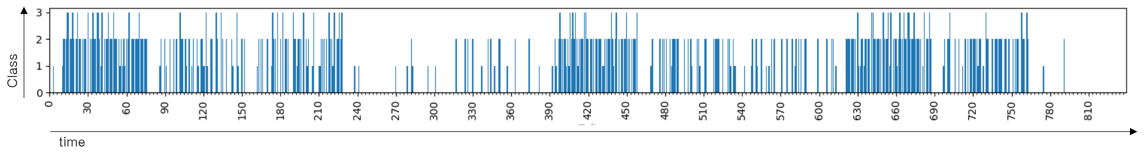

Figure 5: Classification of CPU Usage over time

In Figure 5 on page 9 the classifications are not constant, since the CPUs usage differs from one time step to another. However, if we would consider an average value over an distinct interval (1/11), eleven single classifications can be made:

$$(2, 1, 2, 0, 0, 2, 2, 1, 2, 2, 0)^T$$

This average classification can also be detailed with several intervals of the single phases. How these intervals need to be designed will be evaluated as future work.

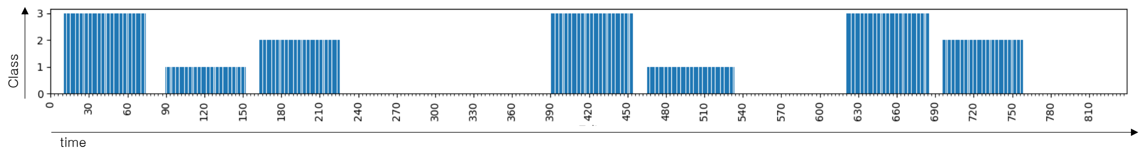

Figure 6: Classification of Memory Usage over time

In contrast to the CPU classification, the memory classifications in Figure 5 on page 9 are constant during each of the eleven phases of the execution:

$$(3, 1, 2, 0, 0, 3, 1, 0, 3, 2, 0)^T$$

Any correlation between this two dimensions need to be investigated statistically and the number of classification levels as well as the intervals themselves will be put to the test. This will include if and how regional usage behavior needs to be considered over the whole application's analysis.

5. Conclusion

In this paper we presented the initial work on the characterization of containerized applications with unknown behavior. The presented fingerprinting approach is still work in progress and its implementation into actual deployment systems. With our analyses of the measured benchmark executions we could demonstrate the potential of this approach. Future work will focus on enhancing the fingerprint generation algorithm, enabling an AI-inspired characterization and elaborate on an efficient and effective mapping function of fingerprints from containerized applications to target systems.

Acknowledgments

This work was supported by the student work of Tobias Celik for his Bachelor's Thesis.

References

[1] M. Höb and D. Kranzlmüller, Enabling easey deployment of containerized applications for future hpc systems, in Computational Science – ICCS 2020, V.V. Krzhizhanovskaya, G. Závodszky, M.H. Lees, J.J. Dongarra, P.M.A. Sloot, S. Brissos et al., eds., (Cham), pp. 206–219, Springer International Publishing, 2020.

[2] B. Gregg, BPF Performance Tools, Addison-Wesley Professional (2019).

[3] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak and G. Carle, Performance implications of packet filtering with linux ebpf, in 2018 30th International Teletraffic Congress (ITC 30), vol. 1, pp. 209–217, IEEE, 2018.

[4] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer et al., The landscape of parallel computing research: A view from berkeley, .