SAND2019-12752C

A Case for Portability and Reproducibility of HPC Containers

R. Shane Canon* and Andrew J. Younge†

*Lawrence Berkeley National Laboratory
National Energy Research Scientific Computing Center
Building 59, 1 Cyclotron Rd
Berkeley, CA USA
Email: scanon@lbl.gov

†Sandia National Laboratories
Center for Computing Research
P. O. Box 5800 MS 1319
Albuquerque, NM USA
Email: ajyoung@sandia.gov

***Abstract*—Containerized computing is quickly changing the landscape for the development and deployment of many HPC applications. Containers are able to lower the barrier of entry for emerging workloads to leverage supercomputing resources. However, containers are no silver bullet for deploying HPC software and there are several challenges ahead in which the community must address to ensure container workloads can be reproducible and inter-operable.**

In this paper, we discuss several challenges in utilizing containers for HPC applications and the current approaches used in many HPC container runtimes. These approaches have been proven to enable high-performance execution of containers at scale with the appropriate runtimes. However, the use of these techniques are still ad hoc, test the limits of container workload portability, and several gaps likely remain. We discuss those remaining gaps and propose several potential solutions, including custom container label tagging and runtime hooks as a first step in managing HPC system library complexity.

I. Introduction

Containerized computing has gained adoption in both High Performance Computing (HPC) and scientific computing and is currently being utilized across many different computing centers, academic communities, and even enterprise HPC [1]. This is driven by in part by a promise to increase productivity and flexibility of scientific workloads through enhanced reproducibility and portability. Since the container image can notionally contain all the required libraries and dependencies and the running environment, this provides an abstraction and removes dependencies on the host system.

The promise of reproducibility with containers has the potential to have a profound impact on scientific computing. For example, to support the DOE NNSA's mission of extending the lifetime of the stockpile without underground testing [2], large-scale modeling and simulation applications that incorporate a multitude of physics and engineering models are executed on leadership-class supercomputers [3]. The inherent complexity of simulations and the environmental factors affecting them require studies involving several application runs over long periods of time. The importance of any single simulation may not be known initially, and subsequent analysis, validation, and verification efforts may identify key results that need to be further analyzed or reproduced, potentially years later. Encapsulating applications together with the entire supporting software environment in a containerized workflow could greatly improve traceability and reproducibility of simulations [4] and improve overall system efficiency in HPC.

However, there are still several areas to consider before containers can deliver on these promises. Containerization and OS-level virtualization promises the potential to improve flexibility, portability and reproducibility for developers through the support of user-defined software stacks [5]. Yet the current mechanisms used within industry for achieving portability and reproducibility also can inadvertently sacrifice performance. The containerized system must still match the host architecture and be capable of exploiting specialized hardware in HPC. The current mechanisms for leveraging advanced HPC hardware and related libraries require pulling in features from the host itself. This method can be brittle and threaten long-term reproducibility, effectively moving away from industry container standards for interoperability.

As the name implies, performance is paramount in High Performance Computing, and is is typically achieved by a combination of specialized hardware components, software to utilize such specialized components, and scale. For an application to achieve peak performance, it often needs to be optimized for the architecture and capable of exploiting advanced hardware like accelerators and interconnects. Achieving this specialization in a containerized application while maintaining a notion of portability has proven to be a difficult balance to maintain. Specifically, there are several considerations to be made with containers for HPC:

- HPC applications need to use specialized interconnects & libraries not found or optimized for in base OS packages.
- Typical container solution requires mapping in libraries, which can cause host-to-container incompatibilities.
- There are differing methodologies in container runtimes for incorporating GPUs and accelerators.
- Users may not know if a given container image can be ported to a different HPC system.
- If portability is possible, users may not know what performance implications exist when running a container on a different HPC system.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. DOE National Nuclear Security Administration under contract DE-NA-0003525.

In this paper, we review some these challenges in utilizing containers for HPC applications and current approaches used in many HPC container runtimes. Many of these approaches have been been demonstrated at scale with applications scaling to thousands of nodes and hundreds of thousands of cores. However, the use of these techniques are still ad-hoc and several gaps remain. We discuss those remaining gaps with a detailed focus on the container & host library compatibility. While not complete, we propose a partial solution using OCI-compliant image labels and runtime hooks to resolve HPC-specific runtime issues. Furthermore, we discuss other options that could lead to additional investigation of container-enabled reproducibility in HPC.

II. State of the Practice

Due to the non-standard hardware found on most leadership-class HPC systems, system-specific libraries are needed within a given HPC container image to best utilize such hardware. A method that has been adopted in most of the HPC-centric container runtimes is to use a combination of bind mounts and dynamic linking to inject optimized libraries from the host into the runtime environment. A common example is MPI applications, which require libraries that have been optimized for the target interconnect. It has been standard practice to include optimized MPI libraries for HPC interconnects for over a decade, with examples including CrayMPI, IntelMPI, MVAPICH, and more.

The HPC container runtime will bind mount the optimized version of the libraries into the container at runtime and force the MPI application to use these libraries in exchange for the versions present in the image (such as a default MPICH installation). This can be done by two ways. The first method is to directly bind mount over the library in the image. For our MPI example, this means overlaying libmpi.a and any other necessary libraries. The other method includes bind mounting all libraries into a separate directory (eg: /host and modifying $LD\_LIBRARY\_PATH$ to prepend that directory. Similar techniques can be used for accelerators like GPUs, and rely on dynamic linking [6], which is generally becoming commonplace in HPC. This approach has proven very successful and has been demonstrated to be robust across different architecturally compatible systems and system upgrades. For example, NERSC has images that were built and deployed around 2016 with Shifter that can still be run on the Cori system, despite multiple system upgrades.

While the approach described above is generally adopted by different HPC container runtimes, the method of implementation and the user interface can vary. For example, Shifter provides a module mechanism that allows a site to specify different extensions that can be selected by the user to customize the runtime including MPI and GPU support [7]. This has lead to near-native performance of containerized apps built with commodity MPICH MPI libraries [8]. Singularity [9] also allows the site to specify default bind mounts that are injected into the container with similar near-native results [10].

However, one downside to these bespoke approaches is the community cannot easily reuse an implementation from one runtime to another. Another issue is the user interface and interaction varies between runtimes, sometimes considerably. As a result, the user may need to learn the right combinations of command-line options between different systems, complicating the adoption of containers for HPC workloads. Furthermore, this approach relies on the application binary interface (ABI) compatibility of the library being overloaded. While MPICH, CrayMPI, IntelMPI and other MPI implementations are ABI compatible [11], OpenMPI and some other proprietary MPI implementations do not offer such flexibility.

For OpenMPI [12], [13], the user must either know the build configuration to match the target machine in advance of the build or leverage an exact module of OpenMPI which is compatible between versions, as well as all of OpenMPI's $RPATH$ dependencies. This can be cumbersome for users new to containerization models who would like to simply take advantage of new container methods for their own application. Furthermore, the customization of host-specific libraries in the container can effectively hamper future ability for baseline reproducibility. The container build must stay inline with a particular cluster or host configuration, such as with user-space MOFED drivers for InfiniBand fabrics, for instance. Recently, Mellanox kept MOFED 4.4 and 4.5 drivers ABI compatible between versions, and is seen as a first step for the container community influencing vendor library compatibility through the help of HPC Container Advisory Council. However, there are no long-term guarantees for future driver releases or for new hardware.

There are emerging standards that could play a role in bring some order to this problem. The Open Container Initiative (OCI) is an open standard for container runtimes and images. The runtime includes standards on hooks that can be used to address some of the use cases in HPC. This standard is already being used by the Sarus runtime [14], among others. While the OCI runtime hooks can play a role, there can be assumptions made in individual hooks that may not work with all runtimes. For example, they may make assumptions about container privileges, which do not translate to current unprivileged container runtime models in HPC.

III. Remaining Gaps

While the approaches described above are generally functional today and emerging standardization can address many issues, gaps still remain. For example, there are currently no tools or conventions for how image creators can specify and communicate requirements for their image to the runtime. The runtime cannot easily determine if an image requires or would benefit from injecting libraries to support HPC hardware, such as GPUs or MPI libraries necessary for Exascale container computing.

While the library injection approach works in many situations, there are still potential issues with this approach. When injecting host libraries into the container, they too have dependencies. These dependencies can be as simple as the

```
$ srun -n 1 shifter /app/hello
...
/app/hello: /lib/x86_64-linux-gnu/
libm.so.6: version 'GLIBC_2.23' not found
(required by /opt/udiImage/modules/mpich/
lib64/dep/libquadmath.so.0)
...
```

Fig. 1. Example error for a glibc mismatch

GNU C Library, otherwise referred to as glibc. Glibc is the implementation of the C standard library used throughout GNU/Linux systems which provides core library support and interfaces to kernel features [15]–[17]. However, glibc from the host could be either significantly older or newer than that in container images. This variation in core system libraries can introduce symbol mismatches between the glibc version in the container and the version used to compile the injected libraries.

This is not a theoretical problem with HPC containers, but in fact a real issue today. A recent major OS update on Cray systems included updated MPI libraries and its dependencies that were built against glibc 2.26 from SuSE SLES15. Trying to inject these optimized MPI libraries into older container images, such as Centos7 images with glibc 2.17, would fail in Shifter. An example of the error is shown in Fig. 1. Fortunately, MPI libraries from the previous Cray OS release still work on the updated system and work with the older images. However, future changes in, for example, the interconnect firmware could break the legacy libraries. This points to a potential long-term challenge in trying to maintain backward and forward compatibility for containerized applications across major system upgrades. Furthermore, this problem was on a single machine; experiencing interoperability across multiple systems even of similar configuration is unlikely.

While this example is only a simple instance of container incompatibility, it in fact is likely be repeated many times over the next few years. For instance, there are several new and emerging interconnect standards with libfabrics, UCX, and Portals-compliant interfaces; many of which are likely to be deployed across leadership supercomputing facilities. Each interconnect will likely outlive any given OS distribution release and dependency library mismatches are almost guaranteed over time. Furthermore, GPUs and accelerators are a key architectural solution for improving the performance of HPC applications [18]. Each device will require custom and often proprietary libraries that will need to be managed explicitly between the host and the container. Such accelerator libraries may be even be tied to specific driver and firmware versions which are specific to a particular facility deployment. If we continue to bind-mount and tune the library path between the container images and the host system for performance, it will be at the expense of portability.

IV. Proposed Solutions

It is unlikely we can provide a ubiquitous solution to all of the issues outlined above. Many of these challenges are not even unique to HPC or containers, but instead describe fundamental limitations of Linux-based operating systems which become exacerbated by the use of containers. However, there are several potential approaches that can help address some of the issues, and any solution will likely require close collaboration with the HPC vendor community, runtime developers, and even end users.

A. Custom Image Labels

A mechanism for capturing and communicating image requirements to the container runtime would provide users with a more seamless experience. This would allow the runtime to determine if it can't meet the requirements of the application in advance and also allow it to determine what set of extensions should be enabled. One potential model would be to leverage the existing label capabilities in the OCI image format to capture these types of requirements. This would require establishing some conventions on how to specify these labels. For example, an image could contain labels indicating that the execution requires a certain version of glibc, MPICH support, or CUDA drivers. This would allow the runtime to read these requirements and determine if it can satisfy those requirements prior to execution and to perform any initialization. This same approach could also be used to specify required architectural details such as processor features, such as AVX512, SSE4, or SVE. A similar concept has been proposed by others in the community [19]. Ideally, a common model could be designed which would allow specifying both hard and soft requirements. For instance, glibc compatibility may be a hard requirement, as symbol mismatchs will cause a failure at container launch. A soft requirement could be simply an older or less-capable GPU, or a un-optimized MPI for debugging or reproducibility purposes where performance isn't a concern.

To demonstrate this approach, we implemented a simple prototype that would work with Shifter and tested it with some sample images. The sample images are tagged with $LABEL$ statement as shown in Table I. An example Dockerfile is shown in Figure 2. To demonstrate the runtime behavior, we implemented a Python-based wrapper script that uses Skopeo [20] to retrieve the metadata for the image and extract the labels. These labels are then used to select the appropriate Shifter modules which is Shifter's mechanism for manipulating the container at runtime to support MPI, GPUs and other features. The prototype illustrates that with the labels and runtime integration, the user has to simply add the appropriate labels to the image and the runtime can determine matching features.

B. Backwards Compatible Libraries

Given the proprietary nature of many of HPC's key performance libraries, there is a serious risk of relying on ABI and library version compatibility between any injected libraries and the images. For example, if Intel decided to add a new

TABLE I
EXAMPLE IMAGE LABELS

Label	Values	Comment
org.supercontainers.mpi	{mpich,openmpi}	Required MPI support, ABI compatibility
org.supercontainers.gpu	{cuda,opencl,rocm, etc}	Required GPU library support
org.supercontainers.glibc	Semantic version: XX.YY.Z	Specific version of GLIBC

```
FROM centos:7

LABEL org.supercontainers.mpi=mpich
LABEL org.supercontainers.glibc=2.17

RUN yum -y update && \
    yum -y install gcc make gcc-gfortran \
           gcc-c++ wget curl

RUN B=mpich.org/static/downloads && V=3.2 && \
    wget $B/$V/mpich-$V.tar.gz && \
    tar xf mpich-$V.tar.gz && \
    cd mpich-$V && \
    ./configure && \
    make && \
    make install

ADD helloworld.c /src/helloworld.c

RUN mpicc -o /bin/hello /src/helloworld.c
```

Fig. 2. Example Dockerfile with Labels

features to IntelMPI that required an ABI change, it could no longer be swapped into the container from the host at run time via a bind-mount. Furthermore, mismatched glibc errors and other system library ABIs between containers and host systems demonstrated in Section 3 points to a more challenging issue of forward compatibility.

One method of addressing this issue for HPC vendors provide multiple libraries, each supporting a cross-product of glibc and other sub-library dependency layers. While tenable, the approach of multiple binary builds will eventually become cumbersome for vendors and HPC facilities to provide long-term and could cause eventual performance degradation or delayed advancement in system software libraries. At best, this is as cumbersome as requiring administrators to maintain multiple custom $lmod$ module installations, as is often done for system software on supercomputers today.

Another approach would be for HPC vendors to provide full source code for each library in order to build into the container images directly. This would enable container developers to compile the various libraries specific to and with the tools from their container image. Entire project development teams could provide base container images with pre-compiled libraries for targeted deployments. It would also allow for supercomputing facilities to support several common builds for container binding, without the entire cross-product of build configurations necessary.

While straightforward, this is essentially intractable approach in reality. Many HPC vendors have spent years fine-tuning their MPI implementations and are unwilling to distribute source code. Expecting Cray to provide the full source to build CrayMPI for every container user would be equivalent to Coca Cola divulging the recipe for Diet Coke on the label of each can.

Currently, there is no simple way to detect or correct for ABI or glibc version mismatches before execution. In the example from Figure 1, a previous version of the library build was sufficient to address the issue and provides a potential solution. If the vendors commit to providing builds against a small range of major library versions, this could potentially provide compatibility over the life of a given supercomputer. Still, further investigation is required to identify what range of versions would be required. Do supercomputing facilities require full compatibility from RHEL6 onward? At what point do we allow the vendors and system software developers advance their library environments? Is there something more structured that the Linux community can do collectively to avoid glibc symbol mismatches?

C. Container Compatibility Layer

Another possible method that could be leveraged is to implement a lower-level container runtime library that manages supercomputing-specific device and library mappings between the host and container images. Specifically, a library would first leverage OCI labels like those in Table 1, or provide mechanisms for detecting hardware and gating access within a container. This tool would run a container image introspection scan at initialization to determine image features and requirements (eg: glib version), which would then be mapped to host-level libraries and handle $LD\_PRELOAD$ and bind or overlay mounts to project the correct and matching system software libraries into the container.

Currently, this concept is utilized by Nvidia through the libnvidia-container library [21]. This provides a base mechanism for handling the driver/library matching at container runtime, as well as a convenient CLI which can be called by users or by container runtime implementations. However, such a technology is particular only to Nvidia GPUs and extending to other hardware and libraries would likely require a significant re-write.

The Cloud Native Computing Organization also started to investigate the usage of a compatibility layer for networks via the Container Network Interface (CNI) [22]. CNI is both a specification and set of plugin interfaces and libraries to implement network configurations for Linux containers. CNI is written in Go, is currently under consideration for wider adoption in Kuberetes [23], and has limited usage in latest versions of Singularity. However, CNI has focused on the Linux network namespaces, no HPC interconnect vendors or providers have currently adopted the CNI usage model to date and integration with HPC-centric interconnects outside the scope of TCP/IP is still relatively unknown.

In reality, such a container compatibility layer would be difficult to gain traction. First, all major HPC container runtimes would have to agree to leverage this shared low-level tool. Given the variety of different runtime implementations and dependencies, this alone would be a significant challenge. Second, it would also need at least partial support from various interconnect, GPU, and accelerator providers and vendor to have any impact on future supercomputing systems. Furthermore, this method would still rely on ABI matching between the container image and the host. While such hurdles are significant, a HPC-focused compatibility layer may be the best way to quickly and seamlessly provide some notion of portability across HPC deployments and container runtimes. A container compatibility layer may also help standardize library and dependencies and help minimize the effects of boutique implementations seen today from various interconnect and accelerator vendors at Exascale.

D. System Virtualization

If providing reproducibility is of paramount concern for the HPC community, it may be viable to to leverage other virtualization technologies beyond just OS-virtualization (aka containers), including ISA-level virtualization technologies which support a Virtual Machine (VM) [24], [25]. Unlike containers, VMs provide full system encapsulation including system-level drivers, a full OS kernel, and emulation of other hardware. The OS kernel and driver pairings could be be tightly controlled to ensure backwards or forwards compatibility with the container runtime, depending on the needs of the container image developers. With this model, it is possible to construct a lightweight hypervisor that instantiates a prepared set of configured OS kernel and driver pairings which have been curated across the life of an supercomputing system, such as for each system upgrade. Then, a container image could be invoked atop the custom-configured Linux kernel and underlying hypervisor in userspace. The result of this dual-virtualization method would be that glibc incompatibilities could be avoided due to the particular matching between the glibc in the container and OS kernel versions. Driver to library ABI mismatches could also be avoided in the same way. Furthermore, root-level operation could be prohibited through the use of an un-privileged container runtime, so the hypervisor still provide access to custom interconnects and accelerator hardware or mount a shared filesystem just as a host OS kernel does natively.

While the combinations of containers and VMs may be a viable approach for ensuring reproducibility in HPC, there are hurdles to overcome. While recent VMs and hypervisors have shown near-native performance for HPC applications using HPC interconnects and GPUs [26], their usage in HPC introduces a measurable source of overhead that may adversely affect performance. Significant implementation would be necessary to enforce user-level operation of container images, integration with current HPC systems, and integration with current vendor software stack and base OS configurations. Also, the use of a hypervisor would not solve all problems of interoperability between system libraries and advanced hardware. For instance, updates in hardware firmware for particular interconnects over time could cause issues with outdated drivers provided within VMs. Furthermore, it would remain an open question whether VMs could help in ensuring portability of software stacks any more-so than container runtimes do today.

Within industry, the combination of virtual machines and containers is much more common, largely due to the relative acceptable tradeoff of performance with hypervisors. Today, most containerized workloads on Amazon's AWS cloud run atop EC2 virtual machines. Docker for Mac leverages xhyve, a lightweight hypervisor for OS X based on bhyve [27]. However, the dual-use of virtualization and containerization within the HPC community does not currently exist in production supercomputers. Any potential design would require significant investments in implementation and general backing from the vendor community to be successful, as well as a method to properly pair container images with compatible OS kernels.

V. Discussion

Since the introduction of container-based computing and associated runtimes in HPC, there is a new hope for both portability and reproducibility of user-defined software environments. While bit-wise reproducibility is unlikely, container usage in HPC may help encapsulate critical software for years to come by providing a foundation for traceability, a critical first step towards long-term reproducibility of scientific software. Furthermore, containers can also help expand the horizon of deployed software ecosystems on HPC resources themselves, effectively paving the way for next generation system software supporting data analytics, machine learning, and next-generation simulation capabilities.

However, current HPC container runtime implementations must make a difficult trade off of these features for performance by swapping in specialized libraries at runtime. In this paper, we have outlined the challenge and trade-offs in detail, with current example problems from real-world supercomputing deployments. We also explore several potential methods for solving or elevating container interoperability, including OCI label standardization and associated runtime hooks, consolidated vendor support, a container compatibility layer, and a dual-virtualization approach. Our hope is that all these methods can be investigated cooperatively and concurrently to eventually find a tractable solution to performant reproducibility. The idea and work proposed in this position paper is only in its infancy and more efforts are needed to ensure performance, portability, and some degree of reproducibility in HPC is possible. As software complexity continues to rise along with the dependency on national supercomputing capabilities, the need for key container features becomes evermore important in the quest for Exascale and beyond.

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. DOE National Nuclear Security Administration under contract DE-NA-0003525. We would also like to thank Ron Brightwell and Kevin Pedretti for their insights and conceptual contributions to this manuscript.

References

[1] Z. Mahmood and R. Hill, *Cloud Computing for enterprise architectures*. Springer Science & Business Media, 2011.

[2] V. Reis, R. Hanrahan, and K. Levedahl, "The big science of stockpile stewardship," in *American Institute of Physics Conference Proceedings*, vol. 1898, no. 1. AIP Publishing, 2017, p. 030003.

[3] C. T. Vaughan, D. Dinge, P. Lin, S. D. Hammond, J. Cook, C. R. Trott, A. M. Agelastos, D. M. Pase, R. E. Benner, M. Rajan, R. J. Hoekstra, and K. H. Pierson, "Early Experiences with Trinity-The First Advanced Technology Platform for the ASC Program," Sandia National Labs, USA, Tech. Rep., 2016.

[4] I. Jimenez, C. Maltzahn, A. Moody, K. Mohror, J. Lofstead, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, "The Role of Container Technology in Reproducible Computer Systems Research," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 379–385.

[5] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 275–287.

[6] M. Franz, "Dynamic linking of software components," *Computer*, vol. 30, no. 3, pp. 74–81, 1997.

[7] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for HPC," *Cray User Group*, 2015.

[8] J. Sparks, "HPC Containers in Use," *Cray User Group*, 2017.

[9] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[10] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A Tale of Two Systems: Using containers to deploy HPC applications on supercomputers and clouds," in *IEEE CloudCom*, 2017.

[11] "MPICH ABI Compatibility Initiative," Webpage. [Online]. Available: https://www.mpich.org/abi/

[12] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2004, pp. 97–104.

[13] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A high-performance, heterogeneous MPI," in *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–9.

[14] L. Benedicic, "Sarus - An OCI-compliant container engine for HPC," in *5th High Performance Containers Workshop (HPCW'19*, 2019.

[15] R. Love, *Linux system programming: talking directly to the kernel and C library*. " O'Reilly Media, Inc.", 2013.

[16] S. Loosemore, R. M. Stallman, A. Oram, and R. McGrath, *The GNU C library reference manual*. Free software foundation, 1993.

[17] B. Gough, *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[18] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[19] C. Kniep, "State of Containers," in *High Performance Container Workshop at ISC18*, 2018.

[20] J. Morales, "Promoting container images between registries with Skopeo." [Online]. Available: https://blog.openshift.com/promoting-container-images-between-registries-with-skopeo

[21] Nvidia, "Libnvidia-container," Webpage. [Online]. Available: https://github.com/NVIDIA/libnvidia-container

[22] C. N. C. Organization, "CNI - the Container Network Interface," Webpage. [Online]. Available: https://github.com/containernetworking/cni

[23] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[24] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 125–134.

[25] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, "A comparison of virtualization technologies for HPC," in *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*. IEEE, 2008, pp. 861–868.

[26] J. P. Walters, A. J. Younge, D. I. Kang, K. T. Yao, M. Kang, S. P. Crago, and G. C. Fox, "GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications," in *2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 2014, pp. 636–643.

[27] N. Natu and P. Grehan, "Nested paging in bhyve," *The FreeBSD Project, http://people. freebsd. org/neel/bhyve/bhyvenestedpaging. pdf*, 2014.