SAND2017-12594C

A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds

Andrew J. Younge, Kevin Pedretti, Ryan E. Grant, Ron Brightwell
Center for Computing Research
Sandia National Laboratories*
Email: {ajyoung, ktpedre, regrant, rbbrigh}@sandia.gov

***Abstract*—Containerization, or OS-level virtualization has taken root within the computing industry. However, container utilization and its impact on performance and functionality within High Performance Computing (HPC) is still relatively undefined. This paper investigates the use of containers with advanced supercomputing and HPC system software. With this, we define a model for parallel MPI application DevOps and deployment using containers to enhance development effort and provide container portability from laptop to clouds or supercomputers. In this endeavor, we extend the use of Singularity containers to a Cray XC-series supercomputer. We use the HPCG and IMB benchmarks to investigate potential points of overhead and scalability with containers on a Cray XC30 testbed system. Furthermore, we also deploy the same containers with Docker on Amazon's Elastic Compute Cloud (EC2), and compare against our Cray supercomputer testbed. Our results indicate that Singularity containers operate at native performance when dynamically linking Cray's MPI libraries on a Cray supercomputer testbed, and that while Amazon EC2 may be useful for initial DevOps and testing, scaling HPC applications better fits supercomputing resources like a Cray.**

I. Introduction

Containerization has emerged as a new paradigm for software management within distributed systems. This is due in part to the ability of containers to reduce developer effort and involvement in deploying custom user-defined software ecosystems on a wide array of computing infrastructure. Originating with a number of important additions to the UNIX chroot directive, recent container solutions like Docker [1] provide a new methodology for software development, management, and operations that many of the Internet's most predominate systems use today [2]. Containers enable developers to specify the exact way to construct a working software environment for a given code. This includes the base operating system type, system libraries, environment variables, third party libraries, and even how to compile their application at hand. Within the HPC community, there exists a steady diversification of different hardware and system architectures, coupled with a number of novel software tools for parallel computing as part of the U.S. Department of Energy's Exascale Computing Project [3]. However, system software for such capabilities on extreme-scale supercomputing resources needs to be enhanced to more appropriately support these types of emerging software ecosystems.

Within core algorithms and applications development efforts at Sandia, there is an increased reliance on testing, debugging, and performance portability for mission HPC applications. This is first due to the heterogeneity that has flourished across multiple diverse HPC platforms, from many-core architectures to GPUs. This heterogeneity has led to increased complexity when deploying new software on large-scale HPC platforms, whereby the path from code development to large-scale deployment has grown considerably. Furthermore, development efforts for codes that target extreme-scale supercomputing facilities, such as the Trinity supercomputer [4], are often delayed by long queue wait times that preclude small-scale testing and debugging. This leaves developers looking for small-scale testbeds or external resources for HPC development & operations (DevOps).

Most containerization today is based on Docker and the Dockerfile manifests used to construct container images. However, deploying container technologies like Docker directly for use in an HPC environment is neither trivial nor practical. Running Docker on an HPC system brings forth a myriad of security and usability challenges that are unlikely to be solved by minor changes to the Docker system. While any container solution will add a useful layer of abstraction within the software stack, such abstraction can also introduce overhead. This is a natural trade-off within systems software, and a careful and exacting performance evaluation is needed to discover and quantify these trade-offs with the use of new container abstractions.

This manuscript describes a multi-faceted effort to address the use of containerization for HPC workloads, and makes a number of key contributions in this regard. First, we investigate container technologies and the potential advantages and drawbacks thereof, within an HPC environment, with a review of current options. Next, we explain our selection of Singularity, an HPC container solution, and provide a model for DevOps for Sandia's emerging HPC applications, as well as describe the experiences of deploying a custom system software stack based on Trilinos [5] within a container. We then provide the first known published effort demonstrating Singularity on a Cray XC-series supercomputer. Given this implementation, we conduct an evaluation of the performance and potential overhead with utilizing Singularity containers on a Sandia Cray testbed system, finding that when vendor MPI libraries

*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. Department of Energys National Nuclear Security Administration under contract DE-NA0003525.

are dynamically linked, containers run at native performance. Given the new DevOps model and the ability to deploy parallel computing codes on a range of resources, this is the first paper that makes use of containers to directly compare a Cray XC-series testbed system against running on the Amazon Elastic Compute Cloud (EC2) service to explore each computing resource and the potential limits of containerization.

II. Background

Throughout the history of computing, supporting user-defined environments has often been a desired feature-set on multi-user computing systems. The ability to abstract hardware or software as well as upper-level APIs has had profound effects on the computing industry. As far back as the 1970s, IBM pioneered the use of virtualization on the IBM System 370 [6], however there have been many iterations since. Beginning last decade, the use of binary translation and specialized instructions on x86 CPUs have enabled a new wave of host ISA virtualization whereby users could run specialized virtual machines (VMs) on demand. This effort has spawned into what is now cloud computing, with some of the largest computational systems today running VMs.

While VMs have seen a meteoric rise in the past decade, they are not without limitations. Early virtualization efforts suffered from poor performance and limited hardware flexibility [7], which dampened their use within HPC initially. In 2010, Jackson et al. compared HPC resources to Amazon EC2 instances [8] to illustrate limitations of HPC on public clouds. However many of these limitations have been mitigated [9] and the performance issues have diminished with several virtualization technologies, including Palacios [10] and KVM [11]. Concurrently, there has also emerged the opportunity for system OS-level virtualization, or containerization within the HPC system software spectrum.

Containerization, or OS-level virtualization, introduces a notion whereby all user-level software, code, settings, and environment variables, are packaged in a *container* and can be transported to different endpoints. This approach creates a method of providing user-defined software stacks in a manner that is similar, yet different than virtual machines. With containers, there is no kernel, no device drivers, nor abstracted hardware included. Instead, containers are isolated process, user, and filesystem namespaces that exist on top a single OS kernel, most commonly Linux. Containers, like VM usage in Cloud computing, can multiplex underlying hardware using Linux cgroups to implement resource limits.

Containers rely heavily on the underlying host OS kernel for resource isolation, provisioning, control, security policies, and user interaction. While this approach has caveats, including reduced OS flexibility (only one OS kernel can run) and an increased risk of security exploits, it also allows for a more nimble, high-performing environment. This is because there is no boot sequence for containers, but rather just a simple process clone and system configuration. Furthermore, containers generally operate at near-native speeds when there is only one container on a host [12]. Due to similar user perspectives, there are a number of studies that compare the performance and feature sets of containers and virtual machines [13], [14], [12], [15], often with containers exhibiting near-native performance.

The use of containers has found a recent assent in popularity within industry like VMs and cloud computing of a few years prior. Containers have been found to be useful for packaging and deploying small, lightweight, loosely-coupled services. Often, each container runs as a single process micro-service [16], instead of running entire software suites in a conventional server architecture. For instance, deployed environments often run special web service containers, which are separated from database containers, or logging containers and separate data stores. While these containers are separate, they are often scheduled in a way whereby resources are packed together in dense configurations for collective system efficiency by relying upon Linux cgroups for resource sharing and brokering. The ability of these containers to be created using reusable and extensible configuration manifests, such as Dockerfiles, along with their integration into Git source and version control management, further adds to the appeal of using containers for advanced DevOps [17].

III. Containers for HPC

As containers gain popularity within industry, their applicability towards supercomputing resources is also considered. A careful examination is necessary to evaluate the fit of such system software to determine the components of containers that can have the largest impact on HPC systems in the near future. Current commodity containers could provide the following positive attributes:

- **BYOE** - Bring-Your-Own-Environment. Developers define the operating environment and system libraries in which their application runs.
- **Composability** - Developers explicitly define how their software environment is composed of modular components as container images, enabling reproducible environments that can potentially span different architectures [18].
- **Portability** - Containers can be rebuilt, layered, or shared across multiple different computing systems, potentially from laptops to clouds to advanced supercomputing resources.
- **Version Control Integration** - Containers integrate with revision control systems like Git, including not only build manifests but also with complete container images using container registries like Docker Hub.

While utilizing industry standard open-source container technologies within HPC could be feasible, there are a number of potential caveats that are either incompatible or unnecessary on HPC resources. These include the following attributes:

- **Overhead** - While some minor overhead is acceptable through new levels of abstraction, HPC applications generally cannot incur significant overhead from the deployment or runtime aspects of containers.
- **Micro-Services** - Micro-services container methodology does not apply to HPC workloads. Instead, one application area per node with multiple processes or threads per container is a better fit for HPC usage models.

- **On-node Partitioning** - On-node resource partitioning with cgroups is not yet necessary for most parallel HPC applications. While in-situ workload coupling is possible using containers, it is beyond the scope of this work.
- **Root Operation** - Containers often allow root-level access control to users; however, in supercomputers this is unnecessary and a significant security risk for facilities.
- **Commodity Networking** - Containers and their network control mechanisms are built around commodity networking (TCP/IP). Most supercomputers utilize custom interconnects that use interfaces that bypass the OS kernel for network operations.

Effectively, these two attribute lists act as selection criteria for investigating containers within an HPC ecosystem. Container solutions that can operate within the realm of HPC while incorporating as many positive attributes and simultaneously avoiding negative attributes are likely to be successful technologies for supporting future HPC ecosystems. Due to the popularity of containers and the growing demand for more advanced use cases of HPC resources, we can use these attributes to investigate several potential container solutions for HPC that have been recently created.

1) Docker: It is impossible to discuss containers without first mentioning Docker [1]. Docker is the industry leading method for packaging, creating, and deploying containers on commodity infrastructure. Originally developed atop Linux Containers (LXC) [12], Docker now has its own `libcontainer` package that provides namespaces and cgroups isolation. However, between security concerns, root user operation, and the lack distributed storage integration, Docker is not appropriate for use on HPC resources in its current form. Yet Docker is still useful for individual container development on laptops and workstations whereby images can be ported to other systems, allowing for root system software construction but specialized deployment elsewhere.

2) Shifter: Shifter is one of the first major projects to bring containerization to an advanced supercomputing architecture, specifically on Cray systems [19]. Shifter, developed by NERSC, enables Docker images to be converted and deployed to Cray XC series compute nodes, specifically the Edison and Cori systems [20]. Shifter provides an image gateway service to store converted container images, which can be provisioned via read-only loopback bind mounts backed by a Lustre filesystem. Shifter takes advantage of chroot to provide filesystem isolation while avoiding some of the other potentially convoluted Docker features such as cgroups. Shifter also can inject a statically linked ssh server into the container to minimize the potential for security vulnerabilities and reduce conflicts with Cray's existing management tools. Shifter recently included support for GPUs [21], and has been used to deploy Apache Spark big data workloads [22]. While Shifter is an open-source project, the project to date has largely focused on deployment on Cray supercomputer platforms and incorporates implementation specific characteristics that make porting to separate architectures more difficult.

3) Charliecloud: Charliecloud [23] is an open source implementation of containers developed for the Los Alamos National Laboratory's production HPC clusters. It allows for the conversion of Docker containers to run on HPC clusters. Charliecloud utilizes user namespaces to avoid Docker's mode of running root-level services, and the code base is reportedly very small and concise at under 1000 lines of code. However, long-term stability and portability of the Linux kernel namespaces that are used in Charliecloud is undetermined, largely due to the lack for support in various distributions such as Red Hat. Nevertheless, CharlieCloud does look to provide HPC containers built from Docker with ease, and its small footprint leads to minimally invasive deployments.

4) Singularity: Singularity [24] offers another compelling use of containers specifically crafted for the HPC ecosystem. Singularity was originally developed by Lawrence Berkeley National Laboratory, but is now under Singularity-Ware, LLC. Like many of the other solutions, Singularity provides custom Docker images to be run on HPC compute nodes on demand. Singularity leverages chroot and bind mounts (optionally also OverlayFS) to mount container images and directories, as well as Linux namespaces and mapping of users to any given container without root access. While Singularity's capabilities initially may look similar to other container mechanisms, it also has a number of additional advantages. First, Singularity supports custom image manifest generation through the definition of Singularity containers, as well as the import of Docker containers on demand. Perhaps more importantly, Singularity wraps all images in a single file that provides easy management and sharing of containers, not only by a given user, but potentially also across different resources. This approach alleviates the need for additional image management tools or gateway services, greatly simplifying container deployment. Furthermore, Singularity is a single-package installation and provides a straightforward configuration system.

With the range of options for providing containerization within HPC environments, we chose to use Singularity containers for a number of reasons. Singularity's interoperability enables container portability across a wide range of architectures. Users can build dedicated Singularity containers on Linux workstations and utilize the new SingularityHub service. This approach provides relative independence from Docker, while simultaneously still supporting importing Docker containers directly. This balanced approach allows for additional flexibility with less compromise from a user feature-set standpoint. Singularity's implementation also solves a number of issues when operating with HPC resources. Issues include the omission of cgroups, root level operation, the ability to bind or Overlay mount system libraries or distributed filesystems within containers, and the support of user namespaces that integrate with existing HPC user control mechanisms.

IV. Container Build, Deployment, & DevOps

Building Docker containers with Sandia mission applications on individual laptops, workstations, or servers can enable rapid development, testing, and deployment times. However, many HPC applications implement bulk synchronous parallel algorithms, which necessitate a multi-node deployment for effective development and testing. This situation is complicated by the growing heterogeneity of emerging HPC systems

architectures, including GPUs, interconnects, and new CPU architectures. As such, there is an increased demand on performance portability within algorithms, although DevOps models have yet to catch up.

A. Container Deployment and DevOps

It is often impractical for applications to immediately move to large-scale supercomputing resources for DevOps and/or testing. Due to their high demand, HPC resources often have long batch queues of job request, which can delay development time. For instance, developers may just want to ensure the latest build passes some acceptance tests or evaluate a portion of the code to test completeness. Current solutions include the deployment of smaller testbed systems that are a miniature version of large-scale supercomputers, but dedicated to application development. Another solution is to leverage Cloud infrastructure, potentially both public or private IaaS resources or clusters to do initial DevOps, including nightly build testing or architecture validation. As such, our container deployment and DevOps model, described below, enables increased usability for all of these potential solutions.

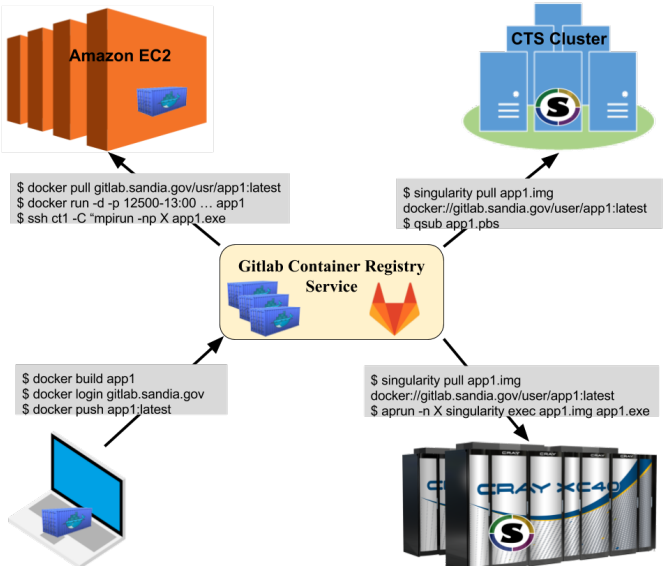

Fig. 1: Deploying Docker Container across multiple HPC systems, including Amazon EC2, a commodity cluster, and a Cray supercomputer

This paper provides a containerized toolchain and deployment process, as illustrated in Figure 1, which architects a methodology for deploying custom containers across a range of computing resources. The deployment initially includes local resources such as a laptop or workstation, often using commodity Docker containers. First, the container Dockerfile and related build files are saved into a Git code project. Due to security concerns surrounding some applications, this container model utilizes an internal Gitlab repository system at Sandia. Gitlab provides an internal repo for code version control, similar in functionality to Github, except that all code is isolated in internal networks. With Gitlab, we can also push entire Docker images into the Gitlab container registry service. While this approach avoids many potential pitfalls when exporting code, this process can also be replicated through the use of Github and Dockerhub for open source projects. Here, developers focus on code changes and package into a local container build using Docker on their laptop, knowing their software can be distributed to HPC resources.

Once a developer has uploaded a container image to the registry, it can then be imported from a number of platforms. While the remainder of this paper focuses on two deployment methods, a Cray XC supercomputer with Singularity and a EC2 Docker deployment, we can also include a standard Linux HPC cluster through Singularity. Initial performance evaluations of Singularity on the SDSD Comet cluster are promising [25]. We expect more Linux clusters to include Singularity, especially given the rise of the OpenHPC software stack that now includes the Singularity container service. As of Singularity 2.3, importing containers from Docker can be done without elevated privileges, which removes the requirement for an intermediary workstation to generate singularity images. Therefore, Docker containers can be imported or pulled from a repository on demand and deployed on a wide array of Singularity deployments, or even left as Docker containers and deployed on commodity systems.

B. Container Builds

Under this DevOps model, users first work with container instances from their own laptops and workstations. Here, commodity container solutions such as Docker can be used, which runs on multiple OSs allowing developers to build code sets targeting Linux on machines not natively running Linux. As Mac or PC workstations are commonplace, the current best practice for localized container development includes the use of Docker, as Singularity is dependent on Linux.

```
FROM ajyounge/dev-tpl
# Load MPICH from TPL image
RUN module load mpi
WORKDIR /opt/trilinos
# Download Trilinos
COPY do-configure /opt/trilinos/
RUN wget -nv https://trilinos.org/...\
/files/trilinos-12.8.1-Source.tar.gz \
-O /opt/trilinos/trilinos.tar.gz
# Extract Trilinos source file
RUN tar xf /opt/trilinos/trilinos.tar.gz
RUN mv /opt/trilinos/trilinos-12.8.1-Source
/opt/trilinos/trilinos
RUN mkdir /opt/trilinos/trilinos-build
# Compile Trilinos
RUN /opt/trilinos/do-configure
RUN cd /opt/trilinos/trilinos-build && make -j 3
# Link in Muelu tutorial
RUN ln -s /opt/trilinos/trilinos-build/pkgs/...\
/opt/muelu-tutorial
WORKDIR /opt/muelu-tutorial
```

An example of a container development effort of interest is introduced by Trilinos [5]. Above is an excerpt that describes an example Trilinos container build. For our purposes, it serves as an educational tutorial for various Trilinos packages, such as Muelu [26]. The Dockerfile above first describes how we inherit from another container `dev-tpl`, which contains Trilinos' necessary third party libraries such as NetCDF and compiler installations, for instance. This Dockerfile specifies

how to pull the latest Trilinos source code and compile Trilinos using a predetermined configure script. While the example below is simply an abbreviated version, the full version enables developers to specify a particular Trilinos version to build as well as a different configure script to build certain features as desired. As such, we maintain the potential portability within the Dockerfile manifest itself, rather than the built image.

While the Trilinos example serves as a real-world container Dockerfile of interest to Sandia, the model itself is more general. To demonstrate this, we will use a specialized container instance capable of detailed system evaluation throughout the rest of the manuscript. This new container includes the HPCG benchmark application [27] to evaluate the efficiency of running the parallel conjugate gradient algorithm, along with the Intel MPI Benchmark suite (IMB) [28] to evaluate interconnect performance. This container image was based on a standard Centos 7 install, and both benchmarks were built using the Intel 2017 Parallel Studio, which includes the latest Intel compilers and Intel MPI library. Furthermore, an SSH server is installed and set as the container's default executable. While SSH is not strictly necessary in cases where Process Management Interface (PMI) compatible batch schedulers are used, it is useful for bootstrapping MPI across a commodity cluster or EC2. As the benchmarks used are open and public, the container has been uploaded to Dockerhub and can be accessed via `docker pull ajyounge/hpcg-container`.

V. Singularity on a Cray

While Singularity addresses a number of issues regarding security, interoperability, and environment portability in an HPC environment, the fact that it is not supported on all supercomputing platforms is a major challenge. Specifically, the Cray XC-series supercomputing platform does not natively support Singularity. These machines represent the extreme scale of HPC systems, as demonstrated by their prevalence at the highest ranks of the Top500 list [20]. However, Cray systems are substantially different from commodity clusters for which Singularity was developed. From a system software perspective, there are several complexities with the Cray Compute Node Linux (CNL) environment [29], which is a highly tuned and optimized Linux OS to reduce noise and jitter when running parallel bulk synchronous workloads, such as MPI applications. The Cray environment also includes highly tuned and optimized scientific and distributed memory libraries specific for HPC applications. On the Cray XC30, this software stack includes a 3.0.101 Linux kernel, an older Linux kernel that lacks numerous features found on more modern Linux distributions, including support for virtualization and extended commodity filesystems.

In order to use Singularity on a Cray supercomputer, a few modifications were necessary to the CNL OS. Specifically, support for loopback devices and the EXT3 filesystem were added into a modified 3.0.101 kernel. A new Cray OS image is uploaded to the boot system and distributed to compute nodes on the machine. These modifications are only possible with root on all compute and service nodes as well as a similar kernel source. This situation means that, for others to use Singularity on a Cray, they too need to make these modifications, or obtain an officially supported CNL OS with the necessary modifications from Cray.

Once Singularity's packages are built and installed in a shared location, configuration is still necessary to produce an optimal runtime environment. Specifically, we configure Singularity to mount /opt/cray, as well as /var/opt/cray for each container instance. While the system currently performs all mount operations with bind mounts, the use of OverlayFS in the future would enable a much more seamless transition and likely improve usability. Otherwise, base directories may be mounted that obfuscate portions of the existing container filesystem, or special mount points must be created. These bind mounts enable Cray's custom software stack to be included directly in running container instances, which can have drastic impacts on overall performance for HPC applications.

As all of Sandia's applications of interest leverage MPI for inter-process communication, utilizing an optimal MPI implementation within containers is a key aspect to a successful deployment. Because containers rely on the same host kernel, the Cray Aries interconnect can be used directly within a container. However, Intel MPI and standard MPICH configurations available during compile-time do not support using the Aries interconnect at runtime. In order to leverage the Aries interconnect as well as advanced shared memory intra-node communication mechanisms, we dynamically link Cray's MPI and associated libraries provided in /opt/cray directly within the container. To accomplish this, a number of Cray libraries need to be added to the link library path within the container. This set includes Cray's uGNI messaging interface, the XPMEM shared memory subsystem, Cray PMI runtime libraries, uDREG registration cache, the application placement scheduler (ALPS), and the configure workload manager. Furthermore, some Intel Parallel Studio libraries are also linked in for the necessary dynamic libraries that were used to build the application. While dynamic linking is handled in Cray's system through `aprun` and Cray's special environment variables, these variables need to be explicitly set within a Singularity container. While this can be done at container runtime or via bashrc by setting `LD_LIBRARY_PATH`, we hope to develop a method for setting environment variables automatically for any and all containers within Singularity, rather than burdening the users with this requirement. Unsurprisingly, the dynamic linking of Cray libraries is also the approach that Shifter takes automatically during the creation of Shifter container images.

This overlay of the Cray MPI environment into containers is possible due to the application binary interface (ABI) compatibility between Cray, MPICH, and Intel MPI implementations. These MPI libraries can be dynamically linked and swapped without impact on the application's functionality. It is worth noting that this is not true with all MPI libraries or other libraries. For instance, OpenMPI uses different internal interfaces and as such is not compatible with Cray's MPI. Future research and standardization efforts are needed to more explicitly define and provide ABI compatibility across systems, as binary compatibility is a key aspect to ensuring performance portability within containers.

VI. Results

The container-based DevOps toolchain described in the previous section provides a model for deploying new applications on HPC platforms. However, we need to explore the consequences of such a system in relation to runtime performance of applications. With this, we look to deploy HPCG and IMB benchmark containers in different ways. The first evaluation case is with a Cray XC series testbed at Sandia built with Singularity. Here, we compare between deploying Singularity containers in multiple configurations against a native, base-case benchmark without any containerization to identify overhead that exists with the proposed system software implementation. Second, because our model now enables these applications to be quickly moved to different resources with relative ease, we can also compare against an Amazon EC2 virtual cluster running Docker.

IMB is used to evaluate a number of network performance aspects. IMB includes point-to-point measurements for best-case network bandwidth and latency, but also embodies a number of critical collective MPI operations, which translate to real-world application communications mechanisms. For many applications of interest to Sandia, an 8-byte global AllReduce operation is of particular importance. While IMB is useful for evaluating detailed network performance, there are also many other factors that can impact application performance. As such, we look to use HPCG to measure aggregate system performance. While Linpack results have previously been a key metric, HPCG provides a more realistic and corollary metric for many modern sparse scientific computing applications. While HPCG performance can also be effected by network performance, it is strongly influenced by system memory bandwidth. Benchmarks are reported as the average of 10 trials for IMB and 3 trials for HPCG, with negligible run-to-run variance that is therefore not shown.

A. Two Systems: A Cray Supercomputer and Amazon EC2

In order to validate the use of Singularity and determine the overhead of containers from a runtime perspective, we deployed container instances on the Volta testbed. Volta is a Cray XC30 system that is part of the Advanced Systems Technology Testbeds project at Sandia National Laboratories through the NNSA Advanced Simulation and Computing (ASC) program. Volta includes 56 compute nodes packaged in a single enclosure, with each node consisting of two Intel Ivy Bridge E5-2695v2 2.4 GHz processors (24 cores total), 64 GB of memory, and a Cray Aries network interface. Compute nodes do not contain local storage. Shared file system support is provided by NFS I/O servers projected to compute nodes via Cray's proprietary DVS storage infrastructure. Each compute node runs an instance of Cray's CNL OS (ver. 5.2.UP04), which is based on SUSE Linux 11 with a Cray-customized 3.0.101 Linux kernel.

By default, the Cray programming environment can create a statically or dynamically linked binary. While dynamic linking at runtime is relatively commonplace on most environments, it is known to have additional overhead. Conversely, the use of Singularity containers requires libraries to be dynamically linked to leverage vendor-tuned libraries, such as Cray's MPI and XPMEM for ABI compatibility. As such, we run the IMB benchmarks natively both with static and dynamic library linking to provide the most in depth comparison, as IMB is sensitive to small performance disturbances.

Given that our existing containers were built with Docker, we also run them on Amazon's EC2 public cloud service. Within EC2, the c3.8xlarge instances are used, which include 2 sockets of 8-core, Intel Xeon "Ivy-Bridge" E5-2680 v2 2.8 GHz processors with hyperthreading, for a total of 16 cores and 32 vCPUs cores per node. While the E5-2680 is normally specified as a 10-core chip, we assume 2 cores per processor are left unscheduled for aiding in Amazon's host management. The c3.8xlarge instance type also includes 60 GB of RAM and 2x320 GB SSDs, and the nodes are connected via a 10 Gb Ethernet network. Each instance was loaded with a RHEL7 image with Docker 1.19 installed. To gain the maximum performance possible from the network, instances were configured to take advantage of SR-IOV, which includes using the `ixgbevf` kernel module to load the Intel 82599 virtual function within the EC2 instance. c3.8xlarge is not directly equivalent to Volta's architecture, with differences including the number of cores per node and operating frequency. However, it provides the closest possible comparison between architectures and both use the same CPU architecture. To keep a close comparison, experiments were run across the same number of MPI ranks and hyperthreading was not used for either architecture. However, the differences in CPUs leads to different socket counts for a given rank. For instance, 768 ranks utilizes 64 sockets and 32 nodes on Volta, 768 ranks uses 96 sockets and 48 nodes on EC2 instances. All resources were requested as single tenancy dedicated host instances in the aws-west-2a zone to mitigate potential resource contention. The c3.8xlarge instances cost $1.68 per hour, coupled with a $2.00 per hour additional cost for dedicated host instances, which brings the total cost of our 48 node virtual cluster to $176.64 per hour on EC2.

B. IMB Interconnect Evaluation

We utilize the IMB benchmark suite first to investigate the maximum available bandwidth between 2 adjacent nodes using the PingPong benchmark. In Figure 2, we can see that peak bandwidth is around 96 Gbps natively on the Aries network. Interestingly, the Singularity container with Cray's MPI library achieves near-native bandwidth at 99.4% efficiency. However, using Intel MPI that was originally used to compile IMB in the container, we see performance drops to around 38 Gbps or just 39.5% of native performance. While Intel MPI is reaching the limits of the Ethernet-over-Aries interconnect it must use, this result immediately highlights the necessity of containers to leverage dynamic linking of vendor libraries to utilize the interconnect natively. For further comparison, we see that the Amazon EC2 network achieves just over 7 Gbps, which represents at best just 6.9% of peak bandwidth in comparison to native Aries. This result demonstrates an order of magnitude difference between commodity cloud infrastructure with 10 Gb Ethernet and a Cray supercomputing resource with a custom HPC interconnect.

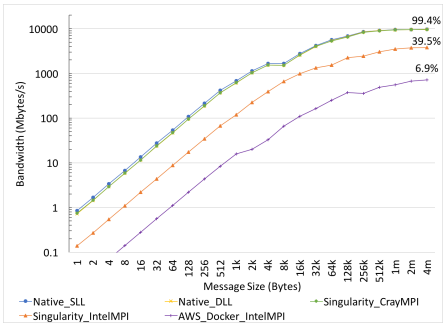

Fig. 2: IMB PingPong Bandwidth

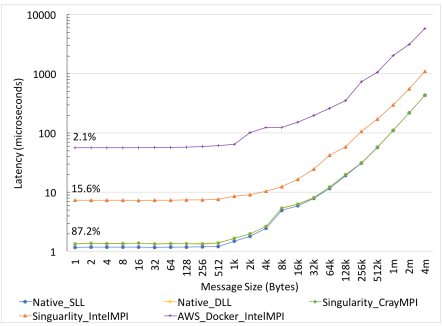

Fig. 3: IMB PingPong Latency

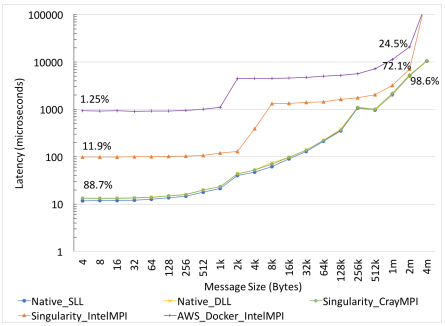

Fig. 4: IMB AllReduce across 768 ranks

Looking next at IMB PingPong latency in Figure 3, the difference between statically linked and dynamically linked libraries becomes noticeable. For small message types, we see an 11.3%, or approximately 200 nanosecond difference between native static and dynamic binaries. While this difference is small, it could have an effect on HPC applications with extreme latency sensitivity, independent of containers. However, when comparing native dynamically linked runs to that of using a Singularity container with Cray MPI, we see no statistically significant difference between the two. This finding illustrates that Singularity containers add no noticeable network latency overhead beyond the requirement for dynamically linked libraries on a Cray XC supercomputer. As with the PingPong bandwidth results, we see that using Intel MPI within the container image has a large impact on performance, with small messages exhibiting a 7-8 microsecond latency, far worse than the 1.3 microsecond latencies with Cray MPI. Overall, across all MPI instances, Volta outperforms the EC2 latency of around 55-60 microseconds with the 10 Gb Ethernet network, which is 49x worse than the native Aries interconnect.

Network performance between Volta XC supercomputing and EC2 does not change when considering MPI AllReduce at scale. From Figure 4, AllReduce was run across 768 ranks spanning 32 nodes on Volta and 48 nodes on EC2 due to CPU differences. When using Singularity containers on Volta, we see there is a small overhead when using dynamically linked libraries with CrayMPI natively, but performance does not differ between dynamically linked native runs and containers. This result confirms that our approach of implementing Singularity does not impact network performance when utilizing the vendor libraries. However, we observe with EC2 that there is a considerable performance impact. At 768 ranks, IMB 8-byte message AllReduce on EC2 is *78x slower* than on the Cray.

C. HPCG Evaluation

To grasp a more complete picture of application performance, we utilize the HPCG benchmark. In Figure 5, HPCG is weak-scaled to a maximum of 768 ranks. We compare HPCG running on Volta natively, within a Singularity container linking Cray MPI, and a Singularity container using Intel MPI. Furthermore, we can leverage our existing research whereby the KVM hypervisor was ported to the same Volta Cray testbed in [11] to also compare containers to a KVM virtual cluster as well as an EC2 virtual cluster.

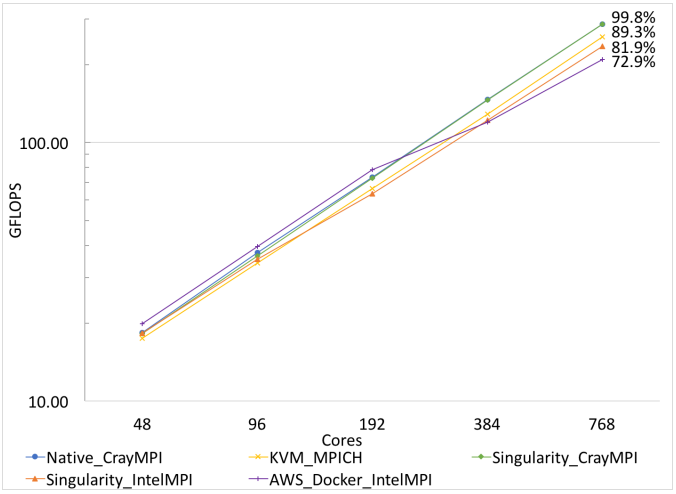

Fig. 5: HPCG weak scaling up to 768 MPI ranks. Volta XC30 running natively, with Singularity using either Cray MPI or Intel MPI libraries, a KVM virtual cluster using MPICH, and AWS EC2 docker containers.

For a single-node Volta HPCG run with 24 ranks, Singularity containers perform within 0.5% of native performance. This is compared to a 3.6% overhead with KVM reported in [11], which demonstrates cumulative CPU, memory, and I/O overhead from virtualization using KVM. Interestingly, we see that for single-node HPCG performance, Amazon EC2 instances (24 ranks on 2 nodes) actually outperforms Volta natively by 15%. While this is surprising at first, EC2 instances have a 15% higher CPU frequency compared to Volta nodes, and network performance is not as important at small node counts.

As HPCG increases in scale, a number of observations emerge as seen in Figure 5. First, we see that the Singularity containers utilizing Cray MPI provide native performance, indicating there is negligible overhead in our implementation. Comparing this to Singularity with Intel MPI, runs can experience an 18.1% overhead, with that overhead likely to increase at further scale. As HPCG running at a larger scale relies more heavily on the interconnect, the overhead with Intel MPI is due to using TCP sockets instead of the native Aries interconnect. Surprisingly, Intel MPI in a Singularity

container has more overhead than running KVM with MPICH, which also uses TCP communication via the Ethernet-over-Aries emulated network. This shows that in fact the selection of an MPI implementation and interconnect utilization has a greater impact on performance than either virtualization or containerization. HPCG overhead at scale becomes even more apparent with Amazon EC2. While a single node HPCG run on EC2 is 15% faster than Volta, any CPU advantage quickly disappears when the EC2 instance count grows beyond 12 nodes. At 768 ranks, our EC2 virtual cluster running Docker incurs a 27.1% overhead compared to native Volta. Furthermore, EC2 has 16.4% more overhead compared to running a KVM virtual cluster on Volta. This indicates that when scaling HPC applications on Amazon's EC2, the overhead is due to the Ethernet network, rather than virtualization.

VII. Conclusion

Given the increased hardware diversity of advanced HPC platforms, together with the software complexity of emerging applications, there is a growing need for containerization within HPC. This paper has evaluated the feasibility of extending HPC system usability through containerization. We have defined a usage model for containerization in HPC and selected Singularity, which to the best of our knowledge is the first published deployment on a Cray system. Then, we evaluated the performance of our containerization methods by comparing native HPC benchmarks to those running in a container. We find that Singularity creates no noticeable overhead when dynamically linking vendor MPI libraries to most efficiently leverage advanced hardware resources. Furthermore, we make a novel comparison of a Cray XC supercomputer to running within Amazon EC2 using our container model and benchmarking apparatus. Our comparison highlights a path for performing initial development, debugging, and testing on cloud resources, such as Amazon EC2, and then seamlessly migrating to a Cray supercomputer for advanced testing and high performance production runs. This new methodology leverages the abundant resource availability in the cloud with the higher performing yet more constrained supercomputing resources to maximize both DevOps efficiency and Cray supercomputer utilization.

References

[1] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[2] Google, "Google container engine," Webpage. [Online]. Available: https://cloud.google.com/container-engine

[3] P. Messina, "Update on the Exascale Computing Project (ECP) ," 2017, HPC User Forum.

[4] K. S. Hemmert, M. W. Glass, S. D. Hammond, R. Hoekstra, M. Rajan, S. Dawson, M. Vigil, D. Grunau, J. Lujan, D. Morton *et al.*, "Trinity: Architecture and early experience," in *Cray Users Group*, 2016.

[5] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps *et al.*, "An overview of the Trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[6] R. Creasy, "The origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, pp. 483–490, 1981.

[7] K. Yelick, S. Coghlan, B. Draney, and R. S. Canon, "The Magellan Report on Cloud Computing for Science," U.S. Department of Energy Office of Science Office of Advanced Scientific Computing Research (ASCR), Tech. Rep., Dec. 2011.

[8] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the Amazon Web Services cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 159–168.

[9] A. J. Younge, J. P. Walters, S. P. Crago, and G. C. Fox, "Supporting high performance molecular dynamics in virtualized clusters using IOMMU, SR-IOV, and GPUdirect," in *Virtual Execution Environments (VEE), in conjunction with ASPLOS*, vol. 50, no. 7. ACM, 2015, pp. 31–38.

[10] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt, "Minimal-overhead virtualization of a large scale supercomputer," in *ACM SIGPLAN Notices*, vol. 46, no. 7. ACM, 2011, pp. 169–180.

[11] A. J. Younge, K. Pedretti, R. E. Grant, B. L. Gaines, and R. Brightwell, "Enabling Diverse Software Stacks on Supercomputers using High Performance Virtual Clusters," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 310–321.

[12] 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 *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 636–643.

[13] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 171–172.

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

[15] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, pp. 105–111, 2014.

[16] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure DevOps," in *Cloud Engineering (IC2E), 2016 IEEE International Conference on*. IEEE, 2016, pp. 202–211.

[17] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, 2016.

[18] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The popper convention: Making reproducible systems evaluation practical," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 1561–1570.

[19] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing Docker for HPC," *Proceedings of the Cray User Group*, 2015.

[20] J. Dongarra, H. Meuer, and E. Strohmaier, "Top 500 supercomputers," website, November 2016.

[21] L. Benedicic, M. Gila, S. Alam, and T. Schulthess, "Opportunities for container environments on Cray XC30 with GPU devices," in *Cray Users Group Conference (CUG16)*, 2016.

[22] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling Spark on HPC systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 97–110.

[23] R. Priedhorsky and T. C. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks," Los Alamos National Laboratory, Tech. Rep., 2016.

[24] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017.

[25] E. Le and D. Paz, "Performance Analysis of Applications using Singularity Container on SDSC Comet," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. ACM, 2017, p. 66.

[26] A. Prokopenko, J. J. Hu, T. A. Wiesner, C. M. Siefert, and R. S. Tuminaro, "Muelu users guide 1.0," *Techical Report SAND2014-18874, Sandia National Laboratories*, 2014.

[27] J. Dongarra, M. A. Heroux, and P. Luszczek, "HPCG benchmark: A new metric for ranking high performance computing systems," *Knoxville, Tennessee*, 2015.

[28] G. Slavova, "Getting started with intel mpi benchmarks 2017," Webpage. [Online]. Available: https://software.intel.com/en-us/articles/intel-mpi-benchmarks

[29] L. Kaplan, "Cray CNL," 2007, FastOS PI Meeting & Workshop.