Shifter: Containers for HPC

Richard Shane Canon
Technology Integration Group
NERSC, Lawrence Berkeley National Laboratory
Berkeley, USA
Email: scanon@lbl.gov

Doug Jacobsen
Computational Systems Group
NERSC, Lawrence Berkeley National Laboratory
Berkeley, USA
Email: dmjacobsen.gov

***Abstract*—Container-based computed is rapidly changing the way software is developed, tested, and deployed. This paper builds on previously presented work on a prototype framework for running containers on HPC platforms. We will present a detailed overview of the design and implementation of Shifter, which in partnership with Cray has extended on the early prototype concepts and is now in production at NERSC. Shifter enables end users to execute containers using images constructed from various methods including the popular Docker-based ecosystem. We will discuss some of the improvements over the initial prototype including an improved image manager, integration with SLURM, integration with the burst buffer, and user controllable volume mounts. In addition, we will discuss lessons learned, performance results, and real-world use cases of Shifter in action. We will also discuss the potential role of containers in scientific and technical computing including how they complement the scientific process. We will conclude with a discussion about the future directions of Shifter.**

***Keywords*-Docker; User Defined Images; Shifter; containers; HPC systems**

I. Introduction

Linux containers are poised to transform how developers deliver software and have the potential to dramatically improve scientific computing. Containers have gained rapid adoption in the commercial and web space, but its adoption in the technical computing and High-Performance Computing (HPC) space has been hampered. In order to unlock the potential of Containers for this space, we have developed Shifter. Shifter aims to deliver the flexibility and productivity of container technology like Docker [1], but in a manner that aligns with the architectural and security constraints that are typical of most HPC centers and other shared resource providers. Shifter builds on lessons learned and previous work such as CHOS [2], MyDock, and User Defined Images [3]. In this paper, we will provide some brief background on containers. Next we will provided an overview of the Shifter architecture and details about its implementation and some of the design choices. We will present benchmark results that illustrate how Shifter can improve performance for some applications. We will conclude with a general discussion of how Shifter including how it can help scientists be more productive including a number of examples where Shifter has already made an impact.

II. Background

Linux containers have gained rapid adoption across the computing space. This revolution has been led by Docker and its growing ecosystem of tools such as Swarm, Compose, Registry, etc. Containers provide much of the flexibility of virtual machines but with much less overhead [4]. While containers have seen the greatest adoption in the enterprise and web space, the scientific community has also recognized the value of containers [5]. Containers have promise to the scientific community for a several reasons.

- Container simplify packaging applications since all of the dependencies and versions can be easily maintained.
- Containers promote transparency since input files like a Dockerfile effectively document how to construct the environment for an application or workflow.
- Containers promote collaboration since containers can be easily shared through repositories like Dockerhub.
- Containers aid in reproducibility, since containers potentially be referenced in publications making it easy for other scientists to replicate results.

However, using standard Docker in many environments especially HPC centers is impractical for a number of reasons. The barriers include security, kernel and architectural constraints, scalability issues, and integration with resource managers and shared resources such as file systems. We will briefly discuss some of these barriers.

Security: The security barriers are primarily due to Docker's lack of fine-grain ACLs and that Docker processes are typically executed as root. Docker's current security model is an all-or-nothing approach. If a user has permissions to run Docker then they effectively have root privileges on the host system. For example, a user with Docker access on a system can volume mount the `/etc` directory and modify the configuration of the host system. Newer features like user namespace may help, but many of the security issues still exist.

Kernel and Architectural Constraints: HPC system are typically optimized for specific workloads such as MPI applications and have special OS requirements to support fast interconnects and parallel file systems. These attributes often make it difficult to run Docker without some modifications. For example, many HPC systems lack a local disk. This makes it difficult although not impossible to run Docker "out of the box". Furthermore, HPC systems typically use older kernel

versions. This is both for stability reasons but also because newer kernels may lack support for parallel file systems such as Lustre. Unfortunately, these older version often lack some of the features that the more recent versions of Docker rely on. For example, user namespaces requires a very modern kernels (3.8 or later).

Scaling: Since each Docker node typically maintains its own image cache, launching a parallel job that uses a new image would result in every node fetching the new image. This leads to serious scaling and start up issues.

Integration: Most HPC systems form an integrated system that includes a resource manager (e.g. Slurm, PBSpro, Torque/Moab) and parallel file systems (e.g. Lustre). User need the flexibilty that Docker provides but it needs to integrate cleanly into the larger system. This means the ability to schedule resources using the Docker environment and the ability to access the parallel file system. Unfortunately, Docker is not designed to easily integrate with most batch systems and providing access to parallel file systems can create data security issues since users effectively have root privileges in the container.

To address these barriers while providing the benefits of Docker, NERSC, in partnership with Cray, has developed Shifter. Shifter leverages key parts of the Docker ecosystem related to image creation and image distribution, but replaces the docker run-time engine. Shifter builds on previous work and experience. A prototype implementation of Shifter was previously presented [3], but significant changes and progress has been made since then. Areas of improvement include: a complete rewrite of the run-time tool (udiRoot) written in C; a complete rewrite of the image gateway that is more extensible and scalable; a number of new features; and improved integration with the SLURM workload manager. We will review these improvements in detail below.

III. Implementation

In this section, we will describe the overall architecture and details about the implementation. Much of the general design was described in our previous work [3]. While that work was a prototype, much of the overall design has remained. However, several of the components have been re-written or redesigned. We will focus our discussion on these changes, but we will briefly describe all of the components for clarity.

A. Architecture

Shifter consists of several major parts. The main components are: command-line user tools, an image gateway, a run-time launcher called "udiRoot", and workload manager integration (WLM) components. The overall architecture can be seen in Figure 1. We will briefly describe each component in more detail. We will follow the general flow of how the pieces interact when a user submits a job.

B. Command-line tools

The user typically interacts with Shifter using two tools `shifterimg` and `shifter`. The later is the front-end to udiRoot and will be discussed below. The `shifterimg` tool is used to pull images into Shifter and query the available images. In the future, it will be extended to allow removing an image and changing access permissions. Under the hood, the tool is basically creating a REST call to the image gateway which is responsible for actually performing the operations and maintaining the image catalog.

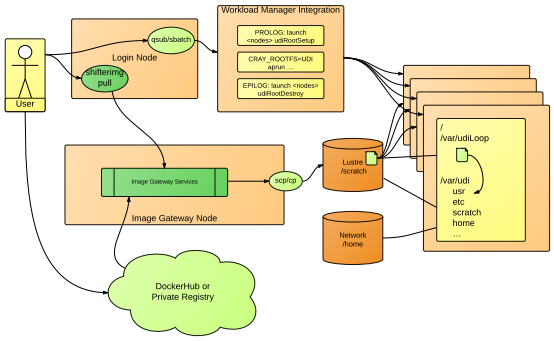

Fig. 1. Diagram of the various components of the Shifter prototype for User Defined Images.

C. Image Gateway

The image gateway has changed significantly compared with the original prototype, however, the overall purposes is the same. The gateway is responsible for importing images from a variety of sources, such as a private Docker regsitry, DockerHub, or a local file system, repacking those images into a format that can be easily consumed by the udiRoot tool, and transporting the packed images to a global file system associated with the target HPC system.

The gateway in the original prototype was a very simple xinetd script that would interact with a locally installed docker engine. The new design is a fully RESTful service written in Python and no longer requires a local Docker Engine instance. Adopting a REST model gives us greater flexibility and makes it much easier to extend and evolve the gateway over time. The new implementation can more easily handle multiple requests, track state, and is more extensible. The new gateway actually consists of several distinct services and introduces several new dependencies. Figure 2 illustrates the architecture of the gateway. The REST interface is implemented using the Flask framework [1]. Flask provides a clean interface for writing Web applications and web services. The flask layer translates requests and uses an underlying Image Manager Layer to service the requests. The image manager layer is the heart of the gateway and is responsible for maintaining the image catalog and dispatching requests. It stores metadata about the available images and operations in a Mongo

[1] http://flask.pocoo.org/

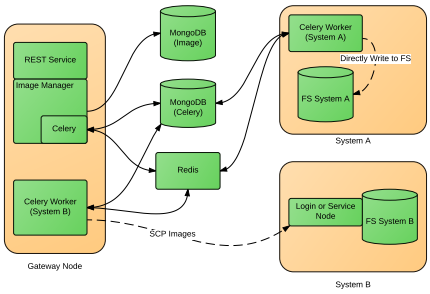

Fig. 2. Diagram of the image gateway and its associated dependencies.

database. For example, when a lookup or listing operation is issued, the image gateway will typically query the Mongo database and return the results in a JSON format. When a pull request is issued, the gateway queues the operation using Celery, an asynchronous task queueing system [6]. Celery provides a framework for queueing tasks that can be serviced by distributed set of workers. Celery supports a number of databases and messaging systems. In a default installation of Shifter, the gateway uses Mongo and Redis to support the Celery queueing system. In addition, a full deployment of Shifter requires a Celery worker for each system (e.g. Cray system or Cluster) supported by the gateway instance. These workers can run on a service node that is a part of the target system and has direct access to the parallel file system (e.g. System A in Figure 2) or can run on the gateway node (e.g. System B in Figure 2). In the first case, this means that the worker can directly write to the system's global file system when it creates an image. In the second case, the worker must copy the image to the target system via an `scp` operation. The first model is typically more efficient, but the second can be useful when meeting the installation requirements for the worker is difficult.

To illustrate the flow of operation, we will describe how a pull request is serviced. In this scenario, the gateway is running on port 5000 of a host called *gateway* and the user is request a Docker image of ubuntu 14.04 for *systema*. The operation starts with a REST call to the Flask API. It looks like the following:

```
POST http://gateway:5000/api/pull/systema/
docker/ubuntu:14.04.
```

The image gateway unpack requests and calls the appropriate underlying method based on the request. The image manager layer then queries MongoDB to see if this image is already present and hasn't been recently fetched. If not, then it queues a request on a queue called "systema" using Celery. Behind the scenes, Celery uses Redis and Mongo to handle control flow and capture results. The request is eventually picked up by the worker servicing the "systema" queue. It then proceeds to pull down the layers for the Ubuntu 14.04 image from DockerHub. Those are saved in a cache so that subsequent requests for the same layer can be skipped. The pulling of these layers is done directly by the worker. It does not need a local installation of the docker-engine to perform these operations. Once all of the layers are downloaded, the worker will unpack the layers to create a copy of the image. This is done on a local temporary space. Once the image is expanded, the work packs the image into the target format (e.g. squashfs or ext4). This effectively "flattens" the layers. If the worker is running directly on the system, it can copy the image into the systems image cache. Alternatively, it will use `scp` to copy the image. The worker returns metadata about the image back through Celery. The image manager layer then creates or updates a record in Mongo. This record includes the status, but also metadata such as the hash associated with the image.

D. udiRoot

The udiRoot component of Shifter is used to instantiate and destroy Shifter containers. A Shifter container has a different scope and meaning from other Linux-container products, owing to Shifter's focus on scalability and HPC use-cases. At its core a Shifter container is a set of manipulations to the Linux Virtual Filesystem Switch (VFS) layer to construct a User Defined Image (UDI) which is capable of supporting chroot'ed process execution. All resource management or resource limits aspects of containers are left to the prevailing Workload Manager running on the system (e.g., SLURM, ALPS, torque, etc). When constructing a Shifter UDI, the udiRoot tools attempt to merge site-policy with the user requested image. This means that udiRoot will manipulate the image to add mount points the site requires, or replace specific files within `/etc`. udiRoot also provides the capacity for a user to request specific volume mounts, even the construction of new temporary spaces, so long as they do not conflict with site policy. Finally udiRoot provides the framework for launching processes into a UDI which merges and translates environment variables coming from the external environment, the desired container environment, and site-required modifications.

udiRoot works by creating a new root filesystem, typically tmpfs mounted on `/var/udiMount`. Within this temporary space all of the base linux needs like `procfs`, `sysfs`, `/dev` are mounted or, in the case of `/dev`, recursively bind-mounted into place. Next some site-defined copies of certain `/etc` files (like `/etc/passwd`, `/etc/group`, `/etc/nsswitch.conf`) are copied into place, as well as bind-mounting site-defined paths into the image (like home directories). Finally the user requested image is mounted to a different path (`/var/udiLoop`), and major components of the user's image are bind mounted into the new root space. This, in effect, gives the site a very flexible way of combining user-defined images with site policy, while retaining strict control over user identities and mappings into the parallel-shared resources copied into the image. A final stage of the UDI preparation process to mount any user-requested volume mounts. Those will be covered in more detail later.

The most common way that udiRoot gets access to the user's requested image is via loop device mounting copy of a read-only squashfs filesystem located on a shared par-

allel filesystem. This means that the user's image is primarily read-only during the execution of the job, however locally mounting the squashfs file has several advantages that greatly enhance process startup times for a variety of workloads. First, because the filesystem is locally mounted, most metadata operations (i.e., where is `/bin/ls` or `/usr/lib64/libpython27.so.2`) occur locally within node memory. Second, because squashfs is compressed data is transferred from the underlying parallel filesystem much more efficiently. Finally, since a single copy of the UDI is used for all MPI ranks in a "fully packed" calculation (assuming WLM integration), the pages are transferred and cached on the node just the first time. All this leads to much more efficient process startup for a dynamically-linked application or an application which reads a number of files in order to launch.

The `shifter` executable is solely responsibile for launching processes into a UDI. This makes `shifter` the strict barrier between the external environment and the UDI environment. This separation (as opposed to allowing other agents to chroot into the UDI), ensures that as few privileged operations as possible are performed within the UDI, and is key to the security of Shifter. Furthermore the `shifter` executable is responsible for merging the environmental differences between the external and UDI environments, to make the transition from the HPC platform into the container as seamless as possible.

Two forms of the UDI exist, one is created in a "global" namespace, meaning that all processes can potentially access the UDI, and the other is created in a private namespace. The global namespace version is useful to allow many different instances of processes to start up within the same UDI. This is beneficial for cases such as many MPI processes starting, since all can read from the same pre-staged UDI. The global namespace UDI are constructed and deconstructed using WLM integration via the setupRoot and unsetupRoot executables. The private namespace UDI is created by the Shifter executable at runtime if either a global UDI is missing or the global UDI does not match the requested UDI. The advantage of the private namespace UDI is that it allows many different images to be used concurrently on the same node or within the same batch job. This capability delivers maximum flexibility to the user.

A recent improvement to the udiRoot component is the restructuring the site and user requestable bind/volume mount capability. The restructuring has enabled Shifter to support new mount options to increase the flexibility and offering of the system. In addition, support for correctly integrating a Shifter container into recent *systemd*-based systems has been achieved. This is primarily by properly marking all Shifter mount points as either "slave" or "private" mode mount propagation. The integration of this feature allows sites to mount or unmount parallel filesystems in the primary environment of a node and have those changes propagate into all functioning Shifter UDIs on that compute node. The recent upgrade in volume mount capabilities also enable support for features like integrating and mounting Cray's DataWarp into a container where the exact paths of the mount points are not known a priori. Perhaps the most exciting and forward-looking enhancement is the addition of XFS-based per-node caches, allowing a user to create large filesystems emulating "local" disk on diskless systems by leveraging the greater capacity and bandwidth of their parallel filesystem. Some usage and performance data for this feature is described in Section IV.

IV. Benchmarking and Comparisons

While the primary motivation for Shifter has been to boost productivity and flexibility of HPC systems by enabling end-users to tailor their environment, Shifter also provides a number of performance benefits as well. Shifter can provide faster start-up time for some applications. With Shifter's new per-node writeable cache, Shifter can address some of the performance penalties for applications that are designed to interact with a local disk.

A. Start-up Time

A common complaint from NERSC users and other large HPC sites has revolved around the poor start-up time for some applications and scripting languages especially Python. The source of this performance issue for Python is primarily due to the poor scalability of the metadata service in Lustre and the way that Python builds up its name space. When a Python application starts up, it must traverse all of the library paths to determine what libraries are present. In addition, when any dynamically linked libraries are loaded, the dynamic shared library loader must traverse the paths defined in `LD_LIBRARY_PATH` to find the libraries. As a result, single python application may need to traverse hundreds or thousands of directories and files to build up and load libraries. This is amplified when the application is scaled up and hundreds or thousands of copies of the python script are launched. On a typical Cray system, these libraries would be stored in Lustre or accessed via DVS to an external file system. There is latency in talking to the remote metadata service associated with the file system and, as the application is scaled, more clients must talk to the same metadata server. This quickly becomes a bottleneck. As a results, even starting a python application at modest scales (thousands of cores) can take hundreds or even thousands of seconds. And this load adversely impacts other users as well. Using faster performing metadata server can help, but any gains can quickly vanish as the application is further scaled.

Shifter helps address this issue because of how the images are stored and accessed by the compute nodes. The images are typically stored in the Lustre file system but are mounted read-only via a loop back mount. This means the kernel can make certain assumptions about what it can safely cache in its buffer cache and directory entry cache.

When a Shifter instance starts, the client node needs to make a single call to the Lustre metadata server to determine the location of the image. Once the image is mounted, the client no longer needs to talk to the Lustre Metadata server to access the contents of the image. When a lookup is performed, the block

holding the metadata must be fetched. But this goes directly to the OST and that block can be safely cached in memory. Subsequent metadata lookups that are in the cached block can avoid talking to the Lustre file system entirely. Furthermore, since the mount is private to that node, the directory entry (dentry) objects can also be safely cached. Consequently, lookups in the same directory can also be accelerated.

A final optimization can be achieved by leveraging the shared library cache (i.e. `ld.so.cache`). Since the user manages the image, can control where libraries are stored, and can modify the `ld.so.conf` that is part of the image, they have the ability to optimize the contents of the `ld.so.cache` during image construction. This means that the loader can avoided traversing the `LD_LIBRARY_PATH` and, instead, directly access and load the requested library. This can dramatically reduce the number of metadata operations required for an application that loads hundreds of shared libraries.

To illustrate these benefits, we ran the Pynamic benchmark [7] on a variety of storage options available on two NERSC Cray systems, Cori and Edison. Edison is 5,576 node Cray XC-30 where each node has two Intel Ivy Bridge processors and 64 GB of RAM. Cori is a Cray XC-40 that is being delivered in multiple phases. Cori currently has 1,630 nodes each with two Intel Haswell processors and 128 GB of RAM. The tested configurations include storing the libraries in the Lustre scratch file system, two different GPFS-based file system (project and common) accessed via Cray's DVS, storing the libraries in Cray's Data Warp solution which uses NVRAM, copying the libraries to a local ramdisk, and, finally, Shifter. The benchmark was run across 4800 nodes with the parameters shown in Table I. Scripts for the run can be found on github [2]. Despite the fact that the Shifter image is backed by the Lustre Scratch, it provides the best overall performance. It even exceeds the ramdisk approach due to the use of the `ld.so.cache` discussed above.

TABLE I
PARAMETERS USED FOR PYNAMIC RUNS.

Parameter	Value
Shared object files	495
Average functions per shared object file	1850
Cross-module calls	enabled
Math library-like utility files	215
Average math library-like utility functions	1850
Additional characters on function names	100

B. Per-node Writable Scratch

Shifter's new per-node writeable-cache space can provide speed up for many applications or frameworks designed to use local disk. The reasons for the speed up are similar to the ones described for the Startup-up improvements. However, the use cases are different. The typical Cray XC-class system lacks a local disk. Consequently the user is left with a small set of options: 1) modify their application to not rely on local disk 2) use the Lustre scratch file system or other external file system 3) use the RAM disk-based `/tmp` space or `/dev/shm` (which is basically equivalent). The first option can require significant work by the user and may not be viable when they are using an existing third-party program. For the second approach, Lustre Scratch can often perform poorly compared with a typical local disk setup if the I/O patterns are small and transactional in nature. Furthermore, as the application scales up, the performance can drop off quickly. Factors of 5x or 10x are sometimes seen in these cases. Data Warp often suffers from the same issues since there is still protocol overhead and data must be transferred from the client to the Data Warp servers. The RAM disk or `/dev/shm` approach typically works much better, but any usage consumes memory which is then not available to the application and is limited to less than the memory size of the node. Furthermore, the user must stage the libraries before starting the applications.

[2]https://github.com/rcthomas/nersc-benchmarks/tree/master/pynamic

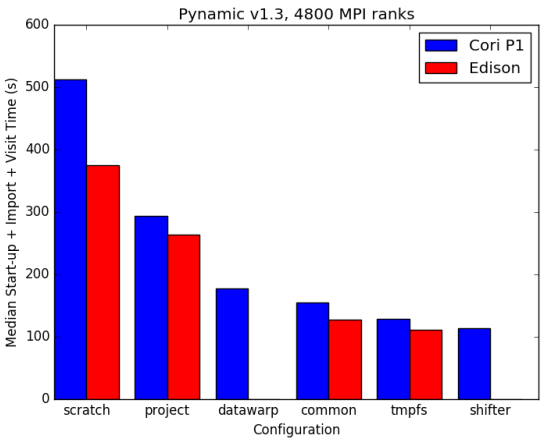

Fig. 3. Comparison of Pynamic execution times on different storage options for Cori and Edison. Courtesy of Rollin Thomas.

To illustrate the impact of this capability, we ran a simple benchmark using the UNIX `dd` command. To simulate an application doing many small writes, we ran the command with a block size of 512 bytes and issued 10,485,760 writes (e.g. `dd if=/dev/zero of=targetfs bs=512 count=10M`). This results in writing out approximately 5.4 GB of data. The results are summarized in Table II. As can be seen, the approach used by Shifter provides between 7x - 10x speed-up, even when the same backing file system is used. Even greater differences have been observed in application tests.

V. DISCUSSION

The goal of NERSC when it designed Shifter was to enable its users to tap into the growing container ecosystem to accelerate their science. As was discussed in the introduction, container computing has the potential to revolutionize how scientific computing is carried out. The ability to share images through DockerHub means that another researcher can easily and quickly replicate published approaches without spending

TABLE II
AVERAGE BANDWIDTH FOR RUNNING A DD TEST WRITING OVER 5 GB OF DATA USING 512 BYTE WRITE OPERATIONS.

Nodes/Threads	Lustre (MB/s)	Shifter (MB/s)
1/1	83	594
10/10	87	625
10/20	67	616
10/40	55	589
20/40	71	627
20/80	55	588

hours or days trying to track down dependencies or struggling with porting issues. It also makes it easier for large collaborations to share tools in a standard, portable way. Shifter allows these potential benefits to be realized without sites having to compromise or risk the security of the systems. We are actively working to promote the adoption of Shifter and containers. The partnership with Cray is intended to ensure that a commercially supported version of Shifter is available to the Cray User community. In addition, we have released Shifter under an open-source modified BSD license and made the code available on GitHub. Other sites have deployed Shifter and even contributed fixes and documentation.

While the primary goal was productivity, Shifter has also provided a number of performance advantages over conventional approaches. Startup times can be 2x to 5x faster. We have even observed orders of magnitude improvements in some cases. Furthermore, the addition of per-node writeable cache finally opens up the possibility to efficiently run applications and frameworks that are designed to work with locally attached storage on systems like the Cray XC-30 that typically lack local disk. Shifter also provides a framework to implement other performance optimizations. For example, if new ways of storing images or writeable temporary spaces are found, they can be implemented into Shifter without requiring major changes by the user.

A. Use Cases

While Shifter has only been in full production since around September, there have already been a number of success stories. We will briefly describe some of the examples.

LCLS: The Linac Coherent Light Source at the Stanford Linear Accelerator (SLAC) is an one of the brightest X-ray sources in existence. It enables scientists to image nanoscale particles and observe at the timescales of chemical reactions. LCLS experiments can generate terabytes of data per day which can create challenges for analysis. Staff and users of the LCLS are working with NERSC to use systems like Cori to do some of their analysis. One challenge in enabling this was getting the PSANA analysis stack developed by LCLS to run on the Cray systems [8]. This software is a complex MPI-enabled python application with many dependencies. Prior to Shifter, staff spent over a month working to port the application. Once they had it ported, they discovered extremely slow startup times at scale due to the issues described above. With the introduction of Shifter, they have been able to easily port their application to a Docker image. This means it is easier to keep the versions running at LCLS and NERSC in sync. These same images can also be used by users to test analysis on their workstation or laptop. Furthermore, Shifter's improved start up performance has helped address the slow start times that were impacting scaling.

LHC Experiments: The Large Hadron Collider (LHC) is the premiere High-Energy Physics facility in the world. Collaborators from across the globe use the facility to understand the most fundamental processes in nature. LHC has a number of detectors with associated collaborations. These projects have very large, complex software stacks that are carefully managed and distributed by the collaborations. However, these tools have been designed to run on clusters that have been specially configured to run their software. Attempts to use HPC style systems for this analysis have had limited success in the past. In many cases, it has been necessary to restrict execution to very specific workloads that have been ported and installed on the HPC system. Staff members from NERSC and other division of Lawrence Berkeley National Lab have used Shifter to demonstrate the ability to package the entire collection of software for several projects into large images that can be used with Shifter. Since these images may contain nearly every package and version of software released by a collaboration, they can be huge (e.g. over 1 TB in size). It has been demonstrated that Shifter can provide nearly constant startup time for these images up to scales of 24,000 cores. ATLAS and ALICE, two detector collaborations at the LHC, have already demonstrated the ability to run workloads on Cori and are making plans to expand this usage in the future. Other similar projects are making similar plans.

LSST: The Large Synaptic Survey Telescope (LSST) is the next-generation telescope currently under construction in Chile and will eventually produce 15 TB of data per night. The collaboration is already engaged in simulations and developing analysis algorithms. The project recognized the potential of Docker containers early on and have already adopted it as a mechanism for distributing their simulation software. Staff members at NERSC working with the collaboration have demonstrated coupling Shifter with workflow software to make it easy to deploy and scale simulations on Cori.

Spark: NERSC has also evaluated using Shifter to support the Spark analysis framework. [9] Spark is an efficient in-memory framework for doing distributed analysis. While Spark can run outside of Shifter, using Shifter's per-node writeable cache space significantly improves the performance and stability of Spark. Spark is designed to work with local disk and uses it to store temporary data including "spills".

B. Handling System Specific Dependencies

One of the promises of container computing and especially Docker is portability. Portability can be straight forward for many applications such as a single node python application that use standard libraries. However, many cases in HPC complicate portability. For example, if the user wants to use Shifter to run an MPI application or a GPU-enabled application, there

are system specific dependencies that the application needs to be synchronized with. For example, the MPI libraries may need to match the version of the firmware running on the interconnect. It is difficult in these circumstances to achieve absolute portability, but there are a few approaches that can be utilized to achieve some measure of portability. We will discuss a few models that are currently being explored. It is worth noting that these different approaches are not exclusive. Different users or sites could potentially adopt different methods.

1) In Image: One obvious model is to have the image contain all of the required libraries and require the image maintainer to keep the image in sync with the target system. This means that any required libraries should be installed into the image (e.g. via an `ADD`, `COPY`, or `RUN` Dockerfile directive). When a system change occurs, the maintainer of the image would be responsible for updating the contents and rebuilding the image to match the system requirements. One benefit of this is everything is captured in the image. So the contents and the run-time environment are well defined. The downside is the image must be rebuilt after any system change. The site can maintain a bundle of libraries that can be easily added to the image to simplify the process of maintaining these images.

2) Managed Base Images: A slightly improved model is for the site or vendor to maintain a standard set of base images. These images would have the appropriate system specific libraries to match the state of the system. For example, the image would contain MPICH or OpenMPI libraries, nVidia libraries, etc. Users would then use Docker's `FROM` directive to base images off of these images. The site could potential deprecate any images that are based off an out-of-date image. Note that Shifter does not currently provide any tools to aid in this detection. The downside to this approach is that whenever a system change occurs, the users would need to rebuild their image to pick up the changes in the base image. In addition, the user is restricted to the set of base images the site or vendor provides. For example, if the user wanted to use an Ubuntu-based distribution but the site only provided RedHat base images, then they would have to use another method.

3) Volume Mounting: Another option is using Shifter's volume mount capability to make system specific libraries available in the Shifter container at run-time. For example, dynamic libraries for MPICH, RDMA libraries, or nVidia libraries can be copied into a special system specific directory and that directory can be mapped at run-time into the Container using a volume mount. The applications in the image would need to be linked and executed in such a way that they would detect and use the libraries. This method can help avoid the need for image rebuilds after upgrades. One downside to this approach, is detail about the run-time environment are maintained outside of the image which means there is some loss in provenance.

4) Better Vendor Support: This problem could be simplified by vendors maintaining some level of backwards compatibility in their libraries and firmware. In addition, ideally the libraries or the firmware should detect when there is an incompatibilities and fail gracefully. Ideally, an MPI library should continue to function with newer firmware until critical interfaces or data structures have changed that break backwards compatibility. This would require additional testing and validation during the release cycle, but Shifter could actually aid in this process since older versions could be maintained and tested for compatibility. In addition, vendors such as Cray could start to provide standard base images (ideally based on a few of the most popular distributions) as part of their standard release process. This would help ensure that the images are synchronized with what is installed on the system.

VI. Conclusion

Shifter opens up the potential of container computing to the HPC community. The development of Shifter has advanced significantly beyond the prototype that was first demonstrated a year ago. Since that time, parts of Shifter have been majorly re-written and Shifter has been formerly released under a open-source BSD-based license and is available for download via GitHub. Furthermore, we are working to build an active developer community that can contribute to the development of Shifter. Most importantly, there have been multiple examples of users using Shifter to run applications. These users have come from a variety of application domains including x-ray light sources, high-energy physics, nuclear physics, and astronomy. We have also used Shifter in the deployment of the Spark analytics framework where the ability to create per-node writable temporary spaces without a local disk have been key to boosting performance and stability. In addition to NERSC, several other sites have already successfully deployed Shifter in their environment. We are actively promoting the adoption of Shifter throughout the HPC community, since, ultimately, container computing and Shifter increase in value to the HPC user community as it becomes more ubiquitous and available at the places they need to compute.

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.

The authors also wish to acknowledge the technical input and support from Cray staff, especially Dave Henseler, Dean Roe, Martha Dumler, Kitrick Sheets, and Dani Connor. In addition, thanks Miguel Gila of CSCS for contributions to Shifter. Finally, thanks to other NERSC staff including Rollin Thomas for the Pynamic Benchmark results, Lisa Gerhardt for support of the LHC use cases, and Debbie Bard for support of the LSST use cases.

References

[1] "Docker," https://www.docker.com/.

[2] S. Canon and C. Whitney, "Chos, a method for concurrently supporting multiple operating system," *Computing in High Energy Physics and Nuclear Physics*, 2004.

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

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32, 2014.
[5] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
[6] "Celery," http://www.celeryproject.org/.
[7] "Pynamic: The python dynamic benchmark," https://codesign.llnl.gov/pynamic.php.
[8] D. Damiani, M. Dubrovin, I. Gaponenko, W. Kroeger, T. Lane, A. Mitra, C. O'Grady, A. Salnikov, A. Sanchez-Gonzalez, D. Schneider *et al.*, "Linac coherent light source data analysis using psana," *Journal of Applied Crystallography*, vol. 49, no. 2, pp. 672–679, 2016.
[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.