A Layered Approach for Modular Container Construction and Orchestration in HPC Environments

Quincy Wofford
qwofford@lanl.gov
Los Alamos National Laboratory
Los Alamos, New Mexico, USA

Patrick G. Bridges
patrickb@unm.edu
University of New Mexico
Albuquerque, New Mexico, USA

Patrick Widener
patrick.widener@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

ABSTRACT

Large-scale, high-throughput computational science faces an accelerating convergence of software and hardware. Software container-based solutions have become common in cloud-based datacenter environments, and are considered promising tools for addressing heterogeneity and portability concerns. However, container solutions reflect a set of assumptions which complicate their adoption by developers and users of scientific workflow applications. Nor are containers a universal solution for deployment in high-performance computing (HPC) environments which have specialized and vertically integrated scheduling and runtime software stacks. In this paper, we present a container design and deployment approach which uses modular layering to ease the deployment of containers into existing HPC environments. This layered approach allows operating system integrations, support for different communication and performance monitoring libraries, and application code to be defined and interchanged in isolation. We describe in this paper the details of our approach, including specifics about container deployment and orchestration for different HPC scheduling systems. We also describe how this layering method can be used to build containers for two separate applications, each deployed on clusters with different batch schedulers, MPI networking support, and performance monitoring requirements. Our experience indicates that the layered approach is a viable strategy for building applications intended to provide similar behavior across widely varying deployment targets.

CCS CONCEPTS

• **Software and its engineering → Application specific development environments**; *Reusability*; **Software configuration management and version control systems**.

KEYWORDS

scientific computing; high performance computing; user defined software stack; containers; distributed applications

ACM Reference Format:
Quincy Wofford, Patrick G. Bridges, and Patrick Widener. 2021. A Layered Approach for Modular Container Construction and Orchestration in HPC Environments. In *Proceedings of the 11th Workshop on Scientific Cloud Computing (ScienceCloud '21), June 21, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3452370.3466001

ScienceCloud '21, June 21, 2021, Virtual Event, Sweden

ACM ISBN 978-1-4503-8385-1/21/06…$15.00
https://doi.org/10.1145/3452370.3466001

1 INTRODUCTION

Large-scale, high-throughput computational science faces an accelerating convergence of software and hardware capabilities. Scientists and developers of workflow-based solutions must navigate a resulting complex set of heterogeneous hardware and software environments. Software containers have become a popular option in cloud systems for dealing with these issues. They offer the ability create portable software stacks which can be easily deployed in multiple locations and are well-suited to data-parallel problems. At the same time, developers of such workflows are seeking ways in which they can take advantage of the considerable investments in tightly-bound compute and storage capacity which have been made in high-performance computing (HPC) facilities. The ability to use similar container-based approaches in both cloud and HPC environments would be a significant benefit for computational scientists.

Using containers in HPC contexts is not straightforward, however. HPC systems do not provide the same container orchestration mechanisms available in cloud systems; they require that users launch containers through batch scheduling services originally designed to launch applications on bare metal. As a result, HPC container systems often couple a particular container runtime with custom HPC scheduling and launch systems, undermining the cross-system reusability and reproducibility benefits containers were designed to provide. This has historically reduced the container functionality available on these systems; current HPC container systems do not offer the equivalent of container "pods" which bundle services like performance monitoring with the application itself that are available in cloud deployment systems such as Kubernetes [1].

To address these issues, we propose a software environment based on container layering which is portable to a wide range of cloud and HPC systems, maximizes application dependency isolation within the container, and does not sacrifice performance to achieve these goals. This container layering approach creates opportunities for HPC developers with increasingly complex software dependencies. HPC applications benefit from the expertise of many specialists, and require significant coordination effort. A container development environment which enables niche experts to implement their layer in a container stack, and then hand that work off to another specialist for continued development, is useful.

The remainder of this paper describes the design and implementation of a containerization strategy which satisfies these design

goals without imposing major changes to the cloud or HPC systems they run on. It does so by:

- Defining the responsibilities and interfaces of the different layers in the container image stack in a modular, extensible manner;
- Providing layers which allow containers to be easily launched on a range of HPC scheduling and job management systems;
- Enabling the creation of layers to differentially provide system services which may not be broadly available (or desirable) in all data centers;
- Allowing users to balance system performance and container portability considerations in the design of those layers; and
- Increasing the effectiveness of containerizing applications by increasing their reproducibility in HPC environments.

We demonstrate how layering creates interfaces which are portable across both applications and systems, and how this in turn enables reproducible performance experimentation across applications and systems. We also demonstrate the practical application of our design by describing its use in deploying an application designed to measure performance variations across multiple HPC platforms.

The remainder of this paper is organized as follows: Section 2 describes background required to understand the layered container approach; Section 3 describes how the container image stack is defined and built; Section 4 describes complications associated with container orchestration on HPC systems and the methods we use to overcome them; Section 5 demonstrates the system and application portability of our container image stack; Section 7 offers concluding remarks.

2 BACKGROUND

In this section we discuss the suitability of containers for HPC uses and the challenges associated with distributed container launch on batch scheduling systems.

2.1 Containers vs. virtualization

Containers use partial virtualization to enable user defined software stacks which are independent of the operating system on which they run. Containers differ from full virtualization solutions in that they share the host kernel, eliminating the kernel emulation overhead required for full virtualization. They are widely used in industry, where software developers need to test their work on multiple platforms with limited hardware availability. Until recently, containers were most commonly used in data centers or on distributed systems where software developers and administrators run trusted software with privileged access.

2.2 Container runtimes

Docker [2] is a container solution which is most often used to run trusted software from privileged accounts. While Docker does include an unprivileged mode, the required Docker daemon (`dockerd`) is potentially a point of failure and a performance bottleneck. Several container runtimes exist which are suitable for users who must run untrusted workloads. The expanding selection of HPC-friendly container runtimes include: Podman [3], Singularity [4], Charliecloud [5], and Shifter [6]. Each of these solutions provide custom software stacks safely. Podman carries forward efforts by

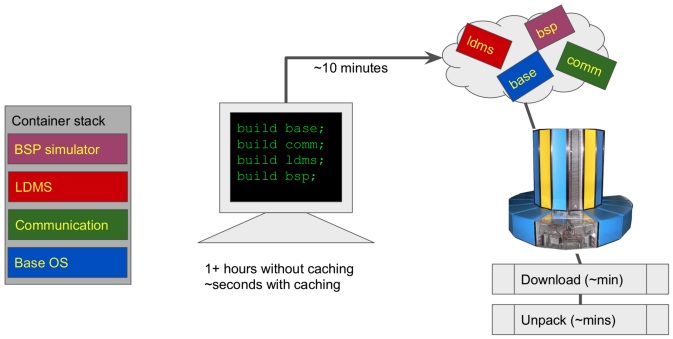

Figure 1: Local development is not cost constrained and allows the user to take advantage of build instruction cacheing. Remote development deploys pre-built container to filesystem, so user can tweak host/container interfaces and runtime configuration.

the Open Container Initiative (OCI) [7] to bring daemonless container runtimes to HPC using OCI-compliant container images. Charliecloud uses a Linux kernel feature called "namespaces" to clone the state of the host system and make changes which only affect the running container. Shifter uses a whitelist of acceptable container images to limit risk, at the cost of total flexibility for the user. Singularity either uses namespace kernel features like Charliecloud, or makes selective use of privileged commands using SETUID runtime binaries. In the latter case, Singularity allows the user to execute privileged commands but only as granted by Singularity. We chose Singularity for this work due to its widespread adoption at the time of writing.

2.3 Local vs. deployed development

A significant advantage of container-based development is the ability to rapidly iterate on application development using a local private system, such as a laptop, and then deploying the same container to a shared system when a desired level of stability has been achieved. Local development is useful when a connection to the target infrastructure is unavailable or when compute time is expensive. Local development uses a container builder that can push images to a container registry (such as Dockerhub for Docker images). Deployed development is useful when troubleshooting the distributed functionality of an application, and Docker's instruction cache greatly reduces the time to make and test changes to a build recipe (Figure 1). Deployed development takes place on the target infrastructure, where deployed containerized scripts can be edited and re-run. The deployed container can be modified for quick tests. Without deployed development mode, even simple changes to the container image require that the entire, potentially hours-long process depicted in Figure 1 take place.

2.4 Research artifacts

The container itself offers a reproducible software stack, and the command script common to Dockerfile-like container specifications can define the series of steps that reproduces an experiment procedure. A user can make simple changes to the host/container

interface script to account for user-specific environment concerns, and re-run the container exactly as it was configured during the data collection phase of a previous experiment. This simple interface allows users to validate previous work with little effort. The container, along with metadata that associates the container with an experiment, can be then be used, distributed, and archived as a research artifact.

3 DEFINING A LAYERED CONTAINER IMAGE

A container is a deployed filesystem tree that a container runtime may reference to execute commands from the container's context. A container recipe is a text file that describes the steps necessary to produce a container. Most container recipes, including Dockerfiles, begin with a line that specifies a base container from which to apply changes. We refer to the practice of layering container recipe files as *container image layering*. In contrast to layered container instruction caching, common among Docker-like build systems, we refer to the layering of containers, not just the instructions a container recipe is composed of.

Each container layer is defined by a container recipe file composed of build instructions for that layer. Building a container from a recipe file takes time, so container layers may also be pre-built and hosted in a container registry to reduce deployment time. If the system where a container is built differs from the deployment target, it's possible to use Spack [8] and ArchSpec [9] to build a container locally which is optimized for the deployment target. These tools place no additional restrictions on the target system.

In some cases, it is not possible to isolate the application from the host operating system. One common example is when proprietary software is provided on the host, but not licensed to the container developer for inclusion in a publicly hosted container. In these situations, the container runtime can bind host-resources at build time, and then again at runtime. Host-matching is an error-prone process that increases development and deployment complexity.

The container stack we demonstrate in this paper is constructed from a series of inherited container layers, beginning with a layer that provides base OS functionality. The next layer in the container image stack is built on this base layer, and subsequent layers are defined by referencing a previous layer. The exact set of layers used is a design issue; we discuss the layers we have used for HPC performance assessment benchmarks and applications in Section 3.2.

Our system uses a Dockerfile-syntax build definition file [10] for its compatibility with many local and cloud-based build systems. We use Docker for build operations. We use the Singularity container runtime, though multiple container solutions exist for HPC which exhibit little or no overhead [11]. We use version control to track container recipes with associated files to provide provenance and reproducibility assurances for our work. We use the Dockerfile `FROM` statement to build containers from sets of layers. When reusing a layer of the image stack, we edit the container image layer definition file to suit the needs of the new layer. When extending the image stack, we create a new Dockerfile using a previous layer as a template. Finally each layer includes a workload script which can be used to drive execution of the container; in our Dockerfile implementation, this script is accessed by the Docker CMD entry.

3.1 Coordinating the host and container

Containers isolate software resources so that the containerized application references containerized dependencies, ignoring those provided by the host operating system except when requested. Maintaining high performance interfaces between the host and the container is a primary concern for all HPC applications. Explicit treatment to maintain high performance interfaces are crucial to any container solution for HPC. There are two ways containers do not isolate from the host: a container can not isolate from the host kernel, and the container must have some interface to the host's job scheduler. Host kernel sharing is an important performance feature, as containers are not required to emulate kernel functionality. Instead, containerized processes are scheduled and executed directly on the host kernel. The host's job scheduler cannot be isolated because this is the component that actually grants the user permission to run on the individual nodes of the distributed system.

We separate the task of coordinating the host and container environment into two elements: the environment of the container in isolation, and the shared host/container environment. This separation of concerns allows container image stack developers to more easily modify and extend previously defined layers. The shared host/container environment acts as a container image stack user interface, and separating this handler simplifies the user experience.

Container environment handling. The container environment handler sets environment variables relevant to the container execution environment that do not depend on host state. For example, these scripts set environment variables such as `LD_PRELOAD` to ensure that container layer libraries are used by other software in the final container. These environment variables must be inherited by each layer that uses them. We use the Docker-style `entrypoint.sh` as the container environment handler. We use `entrypoint.sh` to append container-relevant `PATH` environment settings, and other global variable that are important to our application.

Host/Container Shared Environment Handling. The host/container shared environment handler manages environment attributes which are shared by the host and container and which may change between successive application runs. Each of our container layers have a host/container environment handler called `env.sh`. A container image layer developer will modify the host/container shared environment handler to suit an experimental use case, and a container image layer user will modify the host/container environment handler to manage application input parameters and user-specific host information.

We use `env.sh` to define the elements of the host configuration that must persist to the container. This includes output directories and application input parameters. In our system, the `env.sh` file exists outside of the container, and we copy it into the container at runtime.

3.2 A container implementation in 4 layers

This section describes particular layers of the container image stack developed through this work, as shown in Figure 2.

Base image layer The base image layer is the operating system that will serve as the foundation of the container image

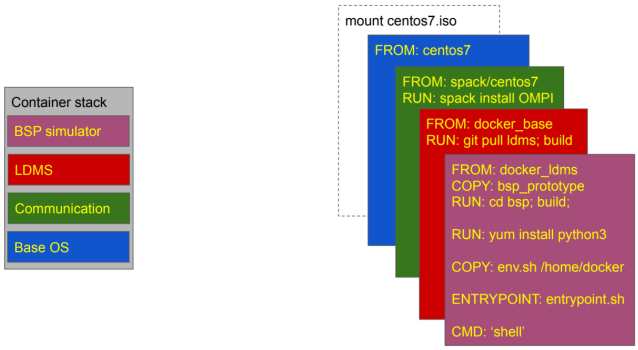

Figure 2: Four container image layers used in benchmarking application container.

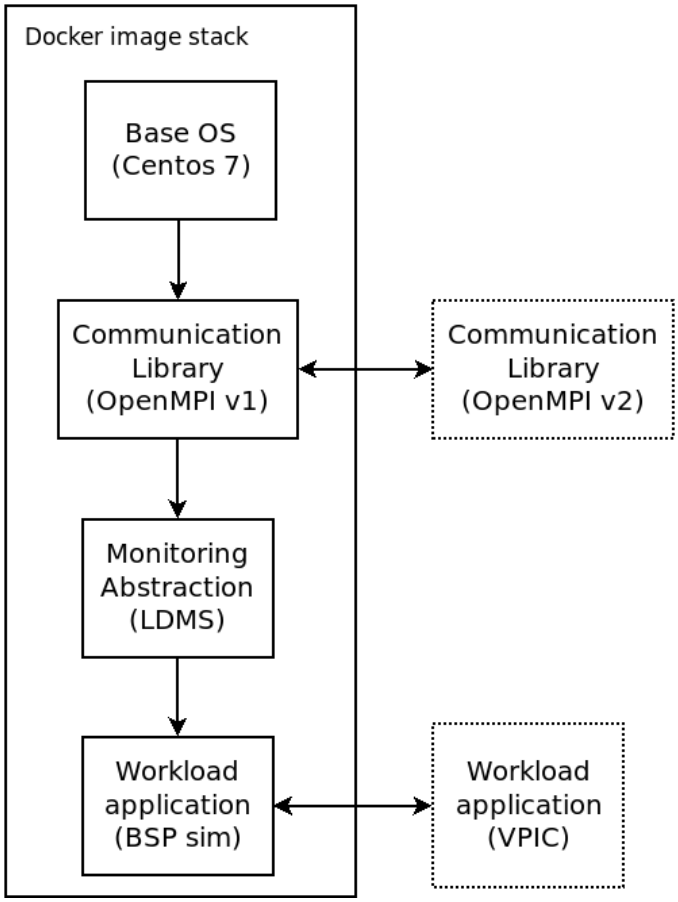

Figure 3: We demonstrate modular container communication and application layers.

stack. The base image layer for our container stack is CentOS 7 [12], as provided by Dockerhub [13]. CentOS is an open source Linux distribution that is notable for its use on many clusters. The base image distribution was chosen to match the host operating system, but matching the host is not necessary.

Communication layer The communication layer is responsible for handling communication across a distributed application during a batch scheduled HPC job. Our communication layer uses an MPI implementation, OpenMPI [14], that includes an orchestration and launch mechanism, `orted` and `mpirun`, respectively. We run our container image stack on two systems, which requires some changes to the build paramaters of the communication layer. We use Spack [8] variants to configure OpenMPI for target systems.

If MPI on the host system is optimized by system administrators, the container's MPI build maybe not be. Similarly, if the container's communication library lacks a PMI interface to enable efficient scheduler orchestration, our launch method may use generic orchestration which fails to take advantage of host-based optimizations. In these situations, it may be preferable to create a communication layer that matches the host's MPI mechanisms. The Singularity documentation refers to this as the "hybrid launch method" [15]. Our method emphasizes portability over potential performance issues by using containerized MPI to orchestrate distributed application launch.

Monitoring layer We anticipate that coupled system and application sampling will be useful for other projects, so we captured this functionality in a monitoring container image layer. The Lightweight Distributed Metric Service (LDMS) [16] provides a practical means to capture the state of every worker node during the execution of a distributed application. LDMS is also capable of measuring calls to the communication library by the distributed application. LDMS works in the client/server paradigm, with system *samplers* acting as clients, and one or more system *aggregators* working as the server.

LDMS is an example of how service complexity can be conveniently expressed using layered container images, with functionality similar to pod-services offered by container orchestration frameworks like Kubernetes [1] and Podman [3]. The term "pod-service" describes services that must run as separate processes alongside a workload application. LDMS runs as a daemon on each sampler node, and the aggregator runs as a separate service on the manager/primary node. Kubernetes and Podman require additional software infrastructure to run, however, and our method of providing a launch example requires no special infrastructure.

Workload application layer The workload application layer is responsible for launching an application for testing or experiment. Our workload application is a bulk synchronous parallel application simulator. A bulk synchronous parallel application is any program that can be broken down into periods of parallelism with cycles of synchronization. Mondragon et al. show that under certain conditions, application run-time can be predicted at scales previously untested [17]. This container image stack validates assertions made by this performance prediction work. To demonstrate application portability of this image layer stack, we constructed a second stack by swapping out the workload application layer for

one which encapsulates a Vector Particle-In-Cell (VPIC) [18] application.

3.3 Benefits of the layered container approach

Deployed, layered image stacks make development and deployment tasks easier in several ways:

Convenience and cost efficiency Containers allow developers to do most of their work on a local computer, where compute time is not charged to a cost account. Working locally also means that a connection to the distributed computer is not required to make progress on a project. Deployed development mode allows users to troubleshoot distributed container launch problems without re-building the container image stack.

Re-usability New layers can substitute for previously created layers without interfering with other stack layers, or they can extend functionality at any desired position within the image stack.

Research artifacts The `CMD` file defines key functionality at each layer of the container image stack, and the container itself resolves dependencies. Together this system can be used as a research artifact with the reproducibility advantages containers provide.

4 DISTRIBUTED APPLICATION LAUNCH

HPC centers rely on batch scheduling systems to assign work to shared systems, and containers were not designed to run on batch scheduled systems. Container deployment, orchestration, and the container runtime context require special consideration in order to run in HPC centers.

4.1 Deployment

We use the Popper [19] experiment workflow manager to manage overall container construction and launch, though a wide variety of scripting and workflow systems could be used for that purpose. This system is responsible for obtaining a job allocation, assembling the container stack from its layers, deploying the container, and running the container on the chosen system. We also use Popper's experiment verification features to validate that container output is reasonable, though such functionality is not necessary in all container deployments.

Our overall experiment pipeline consists of three sequential stages: setup, run, and validate.

setup The purpose of the setup stage is to synchronize a container with the host, and to define an input deck for the application being studied. In this stage the container is retrieved from the Dockerhub container registry and deployed to the NFS filesystem, then the host/container interface script (`env.sh`) is copied from the experiment pipeline home into the deployed container. The user will alter run-time configuration by altering the `env.sh` script.

run The run stage is responsible for running the application workload. The run stage makes requests to a job scheduler from the workflow manager. The run stage requests an interactive allocation, and starts container orchestration with a call to the host's container runtime (Singularity, in our case).

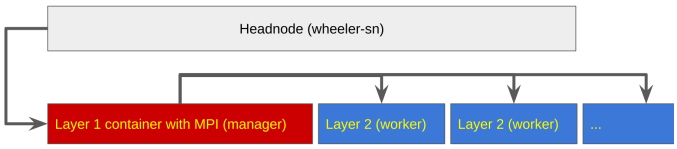

Figure 4: A typical launch of phase 1 and phase 2 containers on an HPC system

Run parameters which affect job launch and runtime behavior were set in the `env.sh` file, described in the setup section. It is possible and useful to store and track the `env.sh` script in the workflow manager, and copy `env.sh` to the mounted container in the run stage. This way, if the host/container shared environment script is not defined on the host and passed to the container, the container will fail. This makes explicit the need for host/container sharing.

validate The validation stage ensures that reasonable outputs exist; the pipeline fails if they do not.

4.2 Container orchestration

One of the design goals of this system is to encapsulate the container environment so that the only host software dependency is the container runtime itself. In order to achieve this isolation goal, a communication library is built for an arbitrary batch job scheduler. In order for orchestration to work with an arbitrary scheduler, the communication library must establish communication paths between nodes that will run the distributed application. Open MPI includes the Open RTE daemon (`orted`), which is suitable for this task.

4.3 Two-phase distributed application launch

The container runtime launch occurs in two phases. This technique deploys the same container twice, but in different ways. This method does not require two deployed containers, but it does require two separate container runtime calls. The first container launch happens on the primary compute node, and puts containerized paths into the environment, including the path to `mpirun`. The second container launch is orchestrated by `mpirun`, and results in a call to containerized `orted` on the remaining compute nodes.

The phase 1 container The phase 1 container is launched on a primary compute node using the host-provided container runtime. The communication library inside the container may be built for a specific job scheduler. This container will call the container-hosted distributed launch mechanism (Open MPI's `mpirun`). The communication library uses a launch agent script for the orchestration mechanism to reference (Open MPI's `orted` references `OMPI_MCA_orte_launch_agent`). The launch agent script loads the host system's container runtime module if it exists, and then uses the container runtime to launch `orted` with all command line arguments.

The phase 2 container The phase 2 container is launched by the communication library's orchestration mechanism (Open MPI's `orted`), and each phase 2 container is a worker node.

Once the phase 2 container launches the orchestration mechanism, it joins the communication network with the manager/primary node and participates in communications as required by application code.

Our workload application, a BSP application simulator, is launched by the container via `commands.sh` with a call to `mpirun`. The communication library "self-orchestrates" the distributed application launch. Although our approach to container orchestration is specific to OpenMPI, similar methods exist for MPICH using environment variable passing via `ssh`. For some system schedulers, such as Slurm, building a communication library with PMI flags is sufficient to enable schedulers to orchestrate a distributed container launch without a self-orchestration tool like `orted`. Our approach uses OpenMPI for the availability of `orted`, which is well suited to porting containers between systems with arbitrary schedulers.

4.4 Open MPI orchestration

The mechanism we use to launch an MPI workload is dependent on the MPI implementation. In Open MPI, when `mpirun` is called, a connection is established to the participating worker nodes, and the service daemon (`orted`) is launched along with a list of derived arguments from `mpirun`. These arguments establish what hardware is used for communication, and other launch configuration details. Ordinarily, each newly connected worker node will search system paths for `orted`. Since our container provides its own communication library, relying on system `orted` would break our distributed container environment.

Open MPI provides a shim that allows us to launch a script rather than the `orted` available in the host system path. The environment variable that must be set is `OMPI_MCA_orted_launch_agent`. This environment variable is set to refer to a script which loads the container environment, just before the `mpirun` command.

When `mpirun` is invoked, the launching process is already in the container environment, so no further action is required on the MPI manager/primary node. However, before Open MPI sets up communication on the worker nodes, `orted` must launch from the container. The script that does this, in this case, is called `ompi_launch.sh`:

```
module load singularity
singularity run bsp_prototype orted $@
```

When singularity invokes the `run` command, `entrypoint.sh` is executed first. The name of the container image referenced above is `bsp_prototype`. The next argument, `orted`, falls through a case statement in the entrypoint script until it matches on the `orted` string, and then launches `orted` from the container context.

The OMPI launch script defines the phase 2 container context, and passes `orted` command line parameters through `entrypoint.sh` to call `orted $@` from the container context in the entrypoint.

5 RESULTS

We demonstrate system portability by running our image stack on the Wheeler cluster at the University of New Mexico's Center for Advanced Research Computing and on the Stampede2 cluster located at the Texas Advanced Computing Center. Wheeler uses a Torque/PBS scheduling system, while Stampede2 uses a Slurm scheduling system. The communication layers of these two container image stacks differ to accommodate the scheduling system.

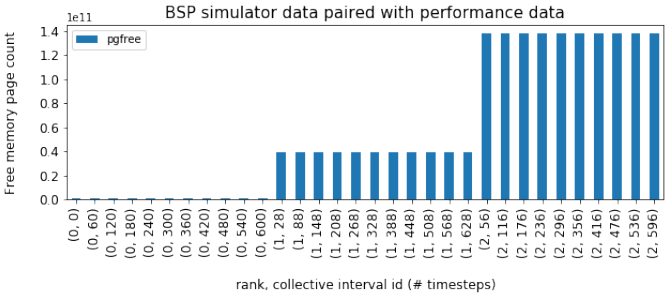

Figure 5: The BSP collective interval is shown paired with system state data, per rank and timestep. Together this demonstrates paired system performance and application data.

All other layers are re-used. In the configuration shown, each BSP simulator sleeps for an amount of time drawn from a Gaussian random variable. The seeds are random, which is why we see some difference between the Wheeler and Stampede2 runs. The sleep times shown in Figure 6 are measured, not obtained from the random variable samples.

We demonstrate application portability of the container layering approach by swapping the BSP simulator application workload layer for a VPIC [18] application workload layer. The layered container approach allowed us to easily port both the complex VPIC application and the LDMS monitoring tools to different systems. Because both VPIC and LDMS are generally difficult and complex to install, this demonstrated that our approach provided a significant improvement in application and monitoring portability.

We demonstrate paired application and system performance data, as collected from the performance monitoring layer of the container image stack in Figure 5. The MPI application was run using one rank per node. Each rank is shown at a particular interval of the BSP simulator, along with the number of free memory pages as reported by the LDMS performance monitoring layer of our container image stack.

6 RELATED WORKS

6.1 User-defined software stacks (UDSS)

Several projects address portable user-defined software stacks for HPC. Ours focuses on providing lean, application-centric containers which require minimal changes to the host OS.

6.1.1 Bare-metal UDSS. Spack [8] was designed to build packages from source on demand. Spack environments define a set of packages and an associated build toolchain. Spack isolates the environment in a user specified unprivileged filesystem root. We use Spack environments to manage the communication library in our container stack; rrecent efforts toward the `spack containerize` script are increasingly useful and interesting to HPC-focused container projects.

6.1.2 Container-based UDSS for HPC. The Extreme-scale Scientific Software Stack (E4S) [20] is a container-based software stack built for exascale machines. E4S containers include a variety of HPC

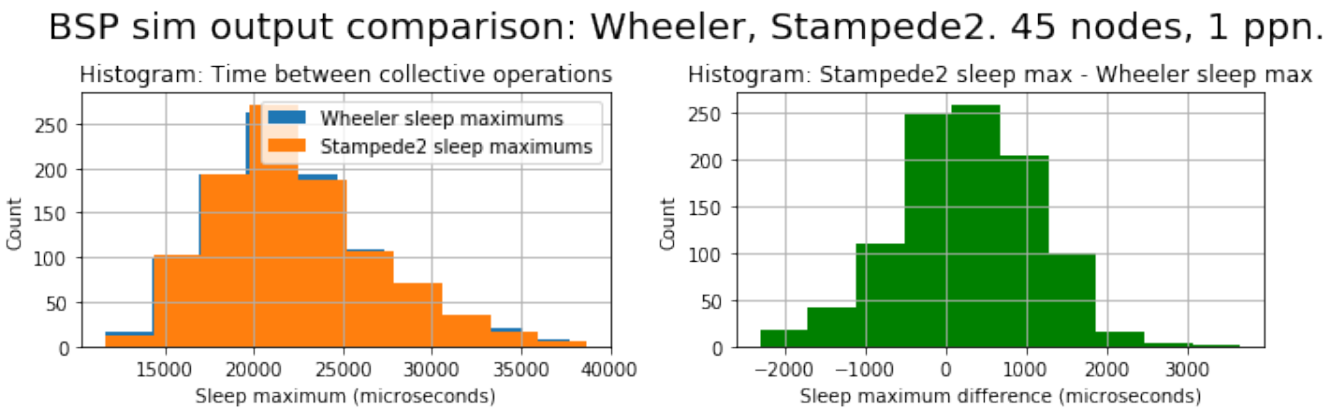

Figure 6: BSP simulator data collected from two systems, where only the communication layer of the required modification to move from a Torque/PBS system to a Slurm system.

applications delivered as a pre-built container, or offered through an E4S-hosted Spack build cache. E4S containers offer pre-built HPC software stacks for target systems and architectures; our method requires a build service and/or a container registry.

6.2 Orchestration systems

Science clouds offer orchestration mechanisms to coordinate virtualization tasks on data center hardware. Configurable hardware with access to full virtualization and high speed filesystem access are the focus of projects such as Bridges [21] and Jetstream [22]. The Chameleon Cloud project [23] provisions high speed network devices along with a configurable amount of virtualization on allocated nodes. However, each of these systems is specially designed and does not support of batch scheduling workloads without substantial effort on the part of the user. In contrast, our approach produces viable containers for unmodified HPC systems.

Sweeney and Thain [24] describe *dynamic* and *static* container orchestration, referring to containers which are deployed once per job run, or once per worker node respectively. In this context, we use a *dynamic* approach to container orchestration. Our containers are deployed to an NFS filesystem once, prior to job submission and before compute node allocation. This dynamic approach is useful when deploying pod services, like our LDMS layer, where separate processes must be launched for LDMS samplers and for the workload application. A static approach would require two container deployments per node: one for LDMS and another for the workload application.

Our approach uses the Open Runtime Environment (`orted`) reference implementation from Open MPI to orchestrate job launch with arbitrary batch schedulers. As long as an Open MPI configuration exists which supports the target scheduler in the `orted` reference implementation, a communication layer may be built for an arbitrary batch scheduler. Increasingly, however, schedulers are performing orchestration through the PMI interface [25]. This affects our system in two ways: orchestration is no longer a role fulfilled by the communication library, and the phase 2 workload launch will need to reference a scheduler specific launch agent rather than a communication library launch agent (e.g. `srun` rather than `mpirun`).

Kubernetes [26] and Docker Swarm [27] are two container deployment systems which have gained widespread adoption in industry for persistent workloads and cloud services, and are gaining traction for HPC workloads as well. Many such systems use Kubernetes systems as a management layer which deploys nodes with job schedulers installed and ready to accept batch workloads. These workloads may themselves be containerized applications. The orchestration mechanism presented in this work does nothing to manage the compute nodes themselves. Our orchestration mechanism is suitable for any target system that supports a container runtime.

Hursey [28] describes a modular approach for MPI components, which facilitates distributed application launch without the use of a specific system scheduler. This approach requires the host to support a generic MPI interface to the system scheduler, and a container that includes an MPI implementation that makes use of the host-supported generic MPI interface.

7 CONCLUSION

In this paper, we presented a design and implementation that achieves modular, composable, and lightweight software stacks which are capable of self-orchestration through Open MPI's `orted`, as provided by the communication layer in the software stack. We defined the responsibilities and interfaces of different container layers in our implementation, we presented modular layers as a tool to balance system performance and container portability, and we demonstrated that this solution is portable across multiple job launch/process management/container orchestration systems. We offer our container recipes as research artifacts which may be referenced to validate this work or re-used to accommodate new application workloads and functionality.

The layered container approach could change the way system administrators and users collaborate to reach meaningful research outcomes. If HPC centers embrace the layered container image approach, the responsibilities of system admins and users would be clearly and separately defined: administrators build base images

for target infrastructures which offer high performance access to hardware, while users are free to implement any software packages they like over the top of the available high-performance base images. Ongoing work toward converged clusters that support batch scheduling systems alongside cloud scheduling systems may also become less complicated if HPC clusters more fully adopt container approaches widely used on cloud platforms.

All components of our container image stack are available for download and referenced in Section 3.2. These container images are also hosted in Dockerhub [13] and available for validation on target systems (Wheeler and Stampede2).

Base image layer The container image stack associated with this layer is managed by its upstream container registry, Dockerhub. The base image layer is available from https://hub.docker.com/r/spack/centos7.

Communication layer The first image layer developed specifically for this work is `docker_base` (available from https://github.com/unm-carc/docker_base). In this git repo, and all associated git repos, there is a branch associated with each target infrastructure. The relevant branches are `carc-wheeler` and `tacc-stampede2`. The image recipe includes a Dockerfile, and Spack environment files.

Monitoring layer The container image stack associated with this layer is available from https://github.com/unm-carc/docker_ldms.

Workload application layer The container image stack corresponding to this layer is available from https://github.com/unm-carc/bsp_prototype.

ACKNOWLEDGEMENTS

Open MPI contributor Josh Hursey provided details about the `OMPI_MCA_orte_launch_agent`, which is the core of the two-phase launch method described in Section 4.3.

This paper was supported in part by the National Science Foundation under Grant No. OAC-1807563. This work was funded in part by Los Alamos National Laboratory, supported by the US Department of Energy contract DE-FC02-06ER25750 (Los Alamos Publication Number LA-UR-20-27116). Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525 (SAND2020-9527C).

REFERENCES

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

[2] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[3] "Podman," podman.io, 2019.

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

[5] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–10.

[6] L. Gerhardt, W. Bhimji, M. Fasel, J. Porter, M. Mustafa, D. Jacobsen, V. Tsulaia, and S. Canon, "Shifter: Containers for hpc," in *J. Phys. Conf. Ser.*, vol. 898, 2017, p. 082021.

[7] "Open Container Initiative - Open Container Initiative," https://opencontainers.org, 2021.

[8] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack package manager: bringing order to HPC software chaos," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[9] M. Culpo, G. Becker, C. E. A. Gutierrez, K. Hoste, and T. Gamblin, "archspec: A library for detecting, labeling, and reasoning about microarchitectures," in *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, 2020, pp. 45–52.

[10] "Dockerfile reference | Docker Documentation," https://docs.docker.com/engine/reference/builder/, 2020.

[11] A. Torrez, T. Randles, and R. Priedhorsky, "Hpc container runtimes have minimal or no performance impact," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, 2019, pp. 37–42.

[12] "Manuals/ReleaseNotes - CentOS Wiki," https://wiki.centos.org/Manuals/ReleaseNotes, 2020.

[13] "qwofford's Profile - Docker Hub," https://hub.docker.com/u/qwofford, 2020.

[14] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[15] "Singularity and MPI Applications," https://sylabs.io/guides/3.4/user-guide/mpi.html, 2020.

[16] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 154–165.

[17] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener, "Understanding performance interference in next-generation HPC systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 384–395.

[18] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.

[19] 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 *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1561–1570.

[20] M. Heroux, J. Willenbring, S. Shende, C. Coti, W. Spear, J. Peyralans, J. Skutnik, and E. Keever, "E4S: Extreme-scale Scientific Software Stack," in *2020 Collegeville Workshop on Scientific Software Whitepapers*, 2020.

[21] N. A. Nystrom, M. J. Levine, R. Z. Roskies, and J. R. Scott, "Bridges: A uniquely flexible hpc resource for new communities and data analytics," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2792745.2792775

[22] C. A. Stewart, T. M. Cockerill, I. Foster, D. Hancock, N. Merchant, E. Skidmore, D. Stanzione, J. Taylor, S. Tuecke, G. Turner, M. Vaughn, and N. I. Gaffney, "Jetstream: A self-provisioned, scalable science and engineering cloud environment," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2792745.2792774

[23] J. Mambretti, J. Chen, and F. Yeh, "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)," in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, 2015, pp. 73–79.

[24] K. M. Sweeney and D. Thain, "Efficient integration of containers into scientific workflows," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, 2018, pp. 1–6.

[25] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt, "Pmix: process management for exascale environments," *Parallel Computing*, vol. 79, pp. 9–29, 2018.

[26] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant, "Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, 2019, pp. 11–20.

[27] N. Nguyen and D. Bein, "Distributed mpi cluster with docker swarm mode," in *2017 ieee 7th annual computing and communication workshop and conference (ccwc)*. IEEE, 2017, pp. 1–7.

[28] J. Hursey, "Design considerations for building and running containerized mpi applications," in *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2020, pp. 35–44.