Docker-Enabled *B*uild and *E*xecution *E*nvironment (BEE): an Encapsulated Environment Enabling HPC Applications Running Everywhere

Abstract—**Variations in High Performance Computing (HPC) system software configurations mean that applications are typically configured and built for specific HPC environments. Building applications can require a significant investment of time and effort for application users and requires application users to have additional technical knowledge. Container technologies like Docker bring great benefits to the application development, build and deployment processes. While much cloud computing infrastructure is already designed to support Docker, little work has been done to support production Docker deployment on HPC systems. In this work, we propose a Docker-enabled Build and Execution Environment (BEE) for HPC systems and detail a standard backend for BEE using virtual machines, the BEE-VM. This brings many of the benefits of Docker to existing HPC machines in user-space without the requirement of specialized pre-installed software and with no system administrator configuration. We show that current HPC application can be easily configured to run within BEE, eliminating the need to reconfigure and rebuild applications for different systems while preserving comparable performance.**

Index Terms—**high performance computing; cloud computing; container.**

I. INTRODUCTION

High Performance Computing (HPC) systems have become critical infrastructure for science and industry. For example, domain experts use HPC systems to run large-scale physical simulations, big data analysis, multi-layer artificial neural networks, molecular dynamics experiments, and DNA sequencing.

As different HPC systems typically have customized software environments, HPC users must often configure and build their application for each specific machine, which is time consuming and can become a bottleneck to productivity. Moreover, in some cases, years or even decades-old legacy applications still serve as key components in the process of scientific or industry research. These legacy applications may have no active support and often require specific deprecated versions of libraries and/or hardware in order to make them run correctly. It may be hard to find or build library implementations that meet the requirement of legacy applications while remaining compatible with current HPC systems. These requirements impose great challenges for HPC users.

In a shared resource computing environment, users commonly implement checkpoint/restoration to stop and resume their computations across allocations, because of time or resource limitations. Differences in software environments may impede the migration of checkpoints across systems that have different software/hardware configurations. A consistent execution environment is also important for HPC application developers. Consistency between developing, testing, and production environments can greatly save developers' time on fixing compatibility issues, which can significantly accelerate the development process.

Virtualized environments, and particularly virtual machines (VMs), have been thoroughly investigated for HPC systems to provide more consistent, isolated, and secure environments [1], [2]. For example, [3] built a virtualized HPC environment using Xen-based VMs. By leveraging high performance I/O passthrough [4], VMs can achieve near-native performance when running HPC benchmarks. In [5], it is shown that current resource managers in HPC systems cannot well supervise VMs and associated critical resources, so they proposed Slurm-V, which extends Slurm with virtualization-oriented capabilities. [6], [7] characterized the performance of running multiple kinds of applications on virtualized HPC clusters. [8] proposed Inter-VM Communication (IVC), a VM-aware communication library to support efficient shared memory communication among computing processes on the same physical host and then they built an MPI library that was IVC enabled.

However, managing application specific VM images is not trivial, given that VM images need to contain files of the operating system, dependent libraries/packages, user applications, and input/output data, which may take several gigabytes of disk space. Migrating images between HPC systems or distributing images among compute nodes can consume a lot of time. On the other hand, Linux container technologies like Docker [9], [10] are more lightweight and yet still provide consistent execution environments for development, build, and deployment. By using Docker, developers only need to build their application once in Docker on their local machine, and then the application can run on any Docker-enabled machine. For doing that, one only needs to pull the Docker images to the target machine. The Docker image only contains minimal operating system composition, application-dependent libraries/packages, and user application, with much less space requirement than VM images [11]. Also, since Docker creates a thin translation layer that allows the guest application to share much of the host operating system, the performance penalty is negligible [12], [13].

It would be of great benefit to bring the advantages of Docker, as realized in the cloud, to HPC users; however, Docker is not usually supported on current HPC systems. The

main reason behind it is that Docker requires a privileged service and a Linux kernel version higher than 3.8, while Linux kernel versions on current HPC production systems are typically far behind this [14]. Especially for national security oriented research facilities, such as Los Alamos National Laboratory, it can take years of security and compatibility evaluation before their production HPC systems can upgrade Linux kernel.

Although HPC systems that run containers are being actively deployed using software such as Shifter [15] and Singularity [16], BEE-VM's approach is complementary. HPC systems that have been built and deployed with Shifter or Singularity can run containerized applications without a VM layer. However, containers must be built or adapted for Singularity and Shifter. BEE-VM brings containerized applications to other HPC machines and, because it runs Docker natively, does not require a rebuild when moving applications across machines.

In this work, we design a Docker-enabled execution framework on HPC with the following design goals:

1) **Automation:** Both the deployment of container-enabled execution environment and the user's application should be done automatically.
2) **Portability:** The execution framework should run across multiple platforms, including HPC systems and cloud computing systems.
3) **Flexibility:** Users should have the flexibility to choose different platforms as their needs change and also migrate application states (via checkpoints) to different platforms.
4) **Reproducibility:** Applications should behave the same across platforms regardless of the host software configurations.
5) **Interfacing Continuous Integration:** The execution framework should support integration with commonly used Continuous Integration (CI) workflows in scientific application development, so that it can effectively free users from handling complicated application compilation.

In this work, we design a new build and execution environment that can enable the execution of Message Passing Interface (MPI) jobs in Docker containers on almost any current HPC system. We call it the Build and Execution Environment (BEE). We specifically detail our standard BEE backend that solves the Linux kernel version issue and provides a more flexible design environment, through a specialized VM containing native Docker, the BEE-VM. For machines that support Kernel-based Virtual Machine (KVM) (on by default in Linux), a hardware accelerated type 1 KVM hypervisor provides bare metal performance. For machines without KVM, we build and configure Quick Emulator (QEMU) in user space. QEMU runs on many host operating systems and on Linux, since kernel 2.6, which makes BEE-VM compatible with almost all HPC machines; however, QEMU without paravirtualization does introduce a significant performance penalty. By using

BOTO API [17], we also provide the BEE backend solution on running MPI jobs in Docker container on cloud platforms. We have a BEE-AWS backend that is able to deploy BEE execution environment on Amazon Web Service (AWS) instances. We also have a BEE-Chameleon, which is able to deploy BEE execution environment on Chameleon Cloud [18]. Most importantly, our solution does not require root/administration privileges of the HPC system, so any user can deploy BEE on HPC systems. Because we use native Docker, users can always benefit from the features of the latest Docker release. The contributions of this work include:

1) **Docker-enabled environment on HPC systems:** By using BEE-VM, we provide Docker-enabled environments into current HPC systems. HPC users can easily Dockerize their application to run on BEE without reconfiguration across different HPC systems. Users can also Dockerize their legacy applications and run on BEE on current HPC systems unmodified.
2) **User space deployment on unmodified HPC systems:** On all QEMU compatible systems, users can deploy BEE-VM on the HPC system they are using without root.
3) **Standard latest Docker support:** We run unmodified Docker daemon inside BEE-VM, so users can Dockerize their application in the standard way. Existing Dockerized application can run on BEE-VM unmodified. Also, Docker inside BEE-VM is upgradeable so that users can always benefit from the latest features of Docker.
4) **Interfacing CI:** Many CI pipelines of current HPC application development projects are already using Docker, so users can run CI-generated Dockerized application on BEE with minimal modification.
5) **Both software and hardware virtualization:** Existing solutions run containers directly on the host, however, different hardware architecture on different hosts may yield different result. With our solution, we provide a hardware virtualization layer, so that together with Docker container we can, if desired, provide a consistent hardware and software environment. This enhances the reproducibility of BEE-VM.
6) **Hybrid HPC and cloud environment:** Besides HPC systems, BEE can also be deployed on cloud computing environments, such as AWS and Chameleon Cloud. We provide the same ability to execute Docker containers to users, so that users can run their HPC application on the cloud without modification.

The rest of this paper is organized as follows: we discuss the BEE framework design in section II and backend BEE-VM in section III. In section IV and V, we introduce the design of BEE-AWS and BEE-Chameleon. Performance is evaluated in section VI. We showcase VPIC, a real HPC application running on BEE-VM in section VII. In section VIII, we discuss other large scale container-enabled execution environments and how BEE is different. Finally, we conclude in section IX.

II. BEE FRAMEWORK DESIGN

`BEE` framework is designed to efficiently organize and manage available computing resources, automatically deploy a Docker-enabled environment, and execute a user's Dockerized application with constant monitoring and scheduling.

As shown in **Algorithm 1**, we outline the general workflow of `BEE` framework. First, users need to provide all the computing resources available. This can be HPC systems or cloud computing systems or both with priorities of using these computing resources. For example, when running a compute intensive application, users may want to prioritize systems with higher computing performance. Then, users need to provide a Dockerized application with an input stack and a configuration file.

Before execution, `BEE` picks the system with the highest priority (**Line 3**). Then, it deploys the Docker-enabled execution environment on the target system. If the execution target environment is HPC system, `BEE-VM` is deployed. If the execution target environment is AWS or Chameleon Clound, then `BEE-AWS` or `BEE-Chameleon` is applied. Both will be discussed in later sections. After the execution environment is deployed, `BEE` checks whether the current application is in an initial stage or needs to restore from a previous checkpoint. If it is in an initial stage, `BEE` attaches the input stack and runs the application as shown in **Line 8 and 10**. Otherwise, `BEE` first migrates previous intermediate data checkpoints from the previous system to the current system (**Line 5**) then attaches the data and restores the checkpoint (**Line 6**) before running.

During execution, `BEE` monitors the current state of the running application. When it is close to the end of the current time slot, `BEE` checks whether the current application has completed. If so, `BEE` outputs the result and stops. Otherwise, if the application has a checkpointing procedure and is indicated by the user, `BEE` initiates the checkpointing procedure and marks $need\_migration$ to ensure the application will resume from the current execution stage for later runs. When finishing the checkpointing procedure, `BEE` checks for the next available system to run. If no other computing systems are available, `BEE` saves the checkpoint to disk, until the user indicates new computing resources are available.

III. BEE-VM DESIGN

Linux container technologies (e.g., Docker) enable consistent software and hardware environments for development, build, and deployment. By using Docker, developers only need to build their application once in a Docker container image on their local machine; then, the dockerized application can run on any Docker-enabled machine. However, Docker is not supported on production HPC systems. Existing containerization solutions on current HPC systems typically require customized HPC environments. The lack of a robust and universal solution to containerization in HPC and the requirement of system customization limits the practicality of container deployments across HPC machines.

To overcome the constraints and provide a more flexible runtime environment, we designed a backend for `BEE`, the

---

Algorithm 1 BEE Framework

Require: $HPC/Cloud_1$: [Host node: H_1, H_2,..., H_k][Time slot: T_1]
Require: $HPC/Cloud_2$: [Host node: H_1, H_2,..., H_k][Time slot: T_2]
...
Require: $HPC/Cloud_m$: [Host node: H_1, H_2,..., H_k][Time slot: T_m]
Require: Computing resource priority list: L
Require: User Dockerized application
Require: Input data: D
Require: User defined hardware configuration: uconf
1: $need\_migration \leftarrow$ False
2: $lastHost \leftarrow$ N/A
3: **while** H $\leftarrow$ get_top(L) **do**
4:    **if** $need\_migration$ is True **then**
5:       data_tranfer($lastHost$, H, D)
6:       restore_data_checkpoint(D, H) // Load data checkpoint into H
7:    **else**
8:       load_initial_data(D, H) // Load initial data into H
9:    **end if**
   // Deploy BEE-VM or BEE-AWS
10:   deploy_bee(H, $user\_application$, $uconf$)
11:   **while** H running normal AND Running-time not close to T **do**
12:     Monitor(H)
13:   **end while**
14:   **if** execution incomplete **then**
15:     pause(H)
16:     $D \leftarrow$ data_checkpoint(H) // Checkpoint application data
17:     $need\_migration \leftarrow$ True
18:     $lastHost \leftarrow$ H
19:     stop(H)
20:   **else**
21:     stop(H)
22:     result $\leftarrow D$
23:     terminate()
24:   **end if**
25: **end while**
26: **if** execution incomplete **then**
27:   $D \leftarrow$ data_checkpoint(H)
28:   stop(H)
29:   store D when no resource available
30: **end if**

BEE-VM, which creates a VM layer on top of the host and then deploys Docker on the VM layer; as shown in **Fig. 1**. Besides providing a standard updated Docker capability to HPC system, additionally the VM layer also brings the following advantages: First, with full control over VM, we can create consistent environments for Docker across platforms. As a result, we can deploy BEE on both HPC and AWS with user applications unmodified. Second, Docker containerization ensures the reproducibility across different software stacks and OSes, while VM guarantees the reproducibility accross different hardware infrastructures, which can be a problem if the container is running directly on the host.
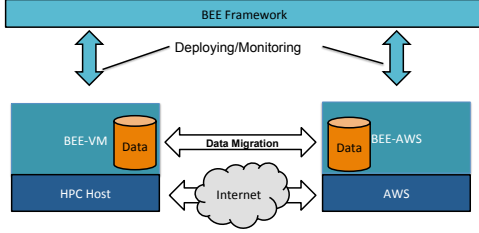

Fig. 1. BEE Framework

The design details of BEE-VM include several parts: network design, storage design, and BEE-VM deployment process. We discuss them in detail in the following sections.

A. Network Design

The network design of BEE-VM mainly targets two functions. First, we need to dynamically configure and deploy our VM and Docker container. This needs to be done automatically and remotely by BEE. We choose to utilize SSH for remote configuration. Also, since we are aiming at HPC application and most HPC applications require MPI, we need to dynamically compose a network for MPI communication between different computing nodes. For enabling that, we create a virtual subnet comprised of all VMs and corresponding Docker containers, so that they can communicate with each other.

1) VM layer: We first discuss the network design at the VM layer. This is necessary because the Docker container will rely on the VM's network configuration for network in the container.

In order to enable the SSH connection to the VM through the host, the hypervisor is configured to use port forwarding to map an unused port on the host to the SSH port on the VM. However, this makes the virtual network interface card (vNIC) hard to utilize by MPI. Although some MPI libraries can be configured to use a designated port and can cooperate with port forwarding, many commonly used MPI libraries (e.g., OpenMPI) use random ports, which is not easy to work with the port forwarding mechanism. So, we created a second vNIC dedicated for MPI communication.

However, allowing application in different node communication to each other through the second vNIC is challenging. The main challenges of designing the network for VMs is that the root privileges are not assigned to regular user but most of the suitable VM networking configurations do require root privileges (e.g., bridging network). To address this limitation, we designed two user space solutions for connecting all second vNICs on HPC systems.

Network Solution 1: Multi-cast In the first solution, we use multi-cast subnet to connect all second vNICs of all VMs together for MPI communication. Since all vNICs are connected to the same subnet, there is no restriction on port usage, so any MPI library can be used. The advantage of this design is that the configuration is simple, and if one node in the multi-cast network failed, it will not affect the connection in between other nodes, which can easily cooperate with a potential fault tolerance mechanism. This network design works best on applications with intensive all-to-all or one-to-all communications, since all packages are naturally sent to all nodes. However, if one-to-one communication is more frequent, the multi-cast network brings high communication overhead, since it generates more data packages in the subnet than necessary.
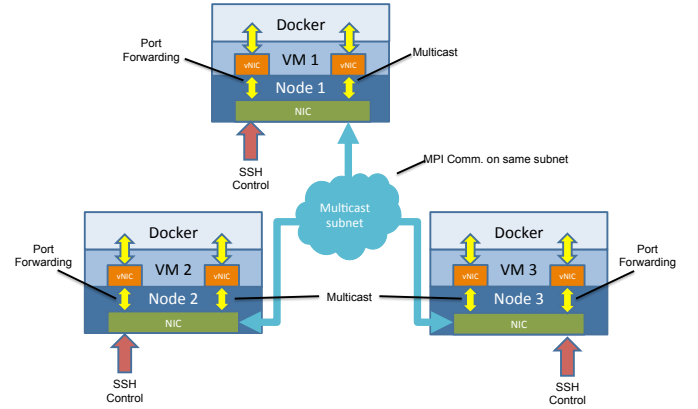

Fig. 2. Network Design using Multi-cast

Network Solution 2: P2P sockets In our second solution, we connect each second vNIC from each VM using point-to-point (P2P) socket connection. This is similar to the wired Ethernet connection between computing nodes in HPC system, except instead of relying on a physical switch we map out a specific routing topology. Since there is a P2P routing path between nodes, there are no unnecessary data packages in the network. However, socket connections between nodes may route via intermediate nodes, so if one node fails, it may break several connected nodes, complicating any potential fault tolerance mechanisms. Also, the performance of this kind of network is affected by the connection pattern between nodes, so we adopt two connection patterns in this solution (more patterns will be studied in the future).

- **Star-shaped connection** In the first approach, we connect nodes using star-shaped topology with the master node in the center. This connection pattern can effectively minimize connection hops between each pair of nodes ($O(1)$). However, since all communication must go through the center node, it may become a hot stop in
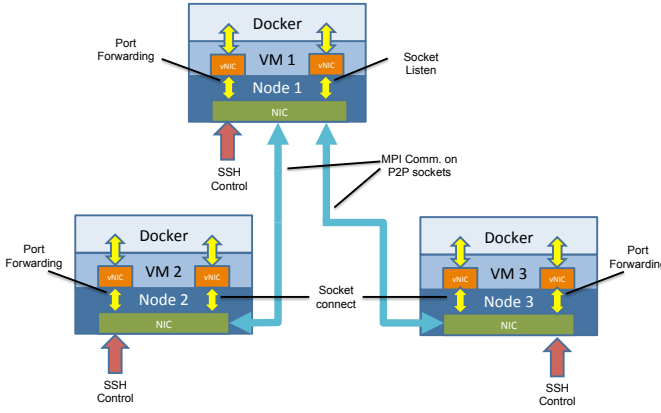
Fig. 3. Network Design using P2P sockets

the network, especially for applications involve intensive network communication or the shared storage design that requires network communication.

- **Tree-shaped connection** The second way to connect all nodes is using tree-shaped topology with the master node as the tree root. In this design, we use a binary tree structure. This connection can mitigate the hot spot issue, but it also increases connection hops between each pair of nodes ($O(n \log n)$).

2) Docker container layer: Based on the network of VM layer, we design the network between Docker containers. There are several network configurations for the Docker containers; however, to minimize network overhead, the most direct configuration is network pass-through. Essentially, all network interfaces on the VM are exposed to the Docker container. Since there is no additional buffer or translation in between, it brings the least overhead. However, it is also considered as the least secure configuration, since everything is exposed. But since we deploy the Docker inside VM, this extra layer already provides enough isolation, so there is no extra security issue here. Since the presented network interfaces are exactly the same between the VM and the Docker container inside it, the software level configuration (e.g., SSH, MPI, etc.) is exactly the same.

B. Storage Design

1) VM layer: In this section, we discuss the design of shared storage system of BEE-VM. General HPC applications usually use some kind of shared file system (e.g., NFS) to share data between processes in real-time. To provide the same environment, we need to build a shared file system across different nodes in BEE-VM. To enable light-weight cross-platform migration, we aim to separate data from the virtualized operating system itself. This separation allows users to easily migrate their data to another platform without transferring heavy operating system files. We design two architectures to allow different processes in different nodes to share files in real-time.

Storage Solution 1: Extra Data Image + NFS In our first solution, we build an extra data image and mount it as the second disk. Due to the copy-on-write characteristic of most virtual images, files updated by one machine are not visible by other machines in real-time. So, we choose to mount this data image only to the master node, and use NFS to share this mounted data disk with other VMs. By using the extra data image, data can be easily migrated. Before computation, input data can be first loaded into the image offline. After execution, the output data is also stored in this image. To move the data to other parts of the user workflow, the user only needs to unmount the data image from the master node of the current BEE cluster and mount it to the master node used for the next stage computation. Also, the image encapsulation can protect data integrity. However, the data image is shared with other worker nodes via NFS, which highly depends on the performance of the virtual network. If the user's application is storage I/O intensive and the network bandwidth is limited, it may consume too much of the network resource, which may degrade MPI communication performance.
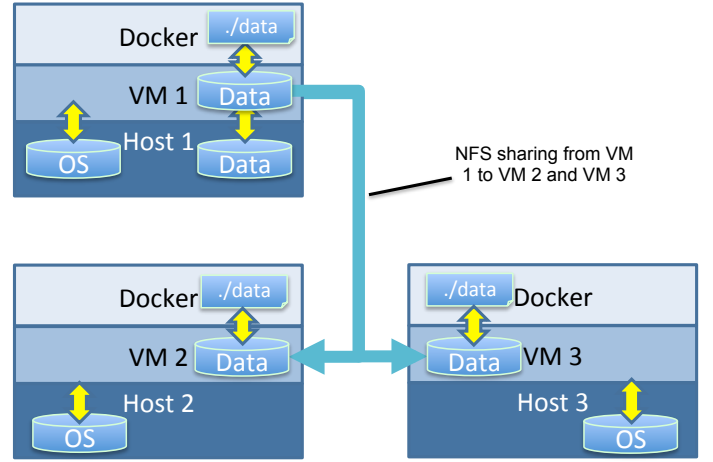

Fig. 4. Shared Storage Design using Extra Data Image + NFS

Storage Solution 2: Virtio In our second solution, we eliminate all the dependency on the virtual network. We use the Virtio feature [19] in QEMU to map a host directory to a directory in VMs. It only requires minimum configuration at VM boot time. Each machine maps the same directory, so the data is shared using file-sharing capability that is commonly used in most HPC systems (e.g., NFS, Lustre, etc.), and it is also visible to the host in real-time. Since it does not rely on the virtual network, the whole virtual network is saved for MPI.

2) Docker container layer: Finally, for data sharing in the Docker layer, we use the data volume mount feature in Docker to mount the shared folder in the VM to a directory in Docker. Since Docker runs as a process at the VM layer, mounting the data volume adds negligible overhead. This configuration is also compatible with all data-shared mechanisms in the VM layer.
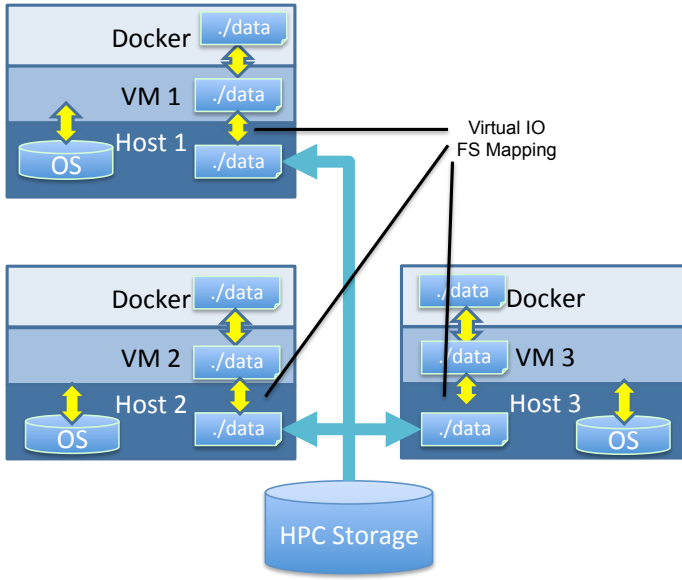
5

Fig. 5. Shared Storage Design using Virtual IO

C. BEE-VM Builder

VM plays an important role in `BEE-VM`. The VM is the core of `BEE-VM` that brings computing resources from host, virtual network, shared storage, user provided Dockerized application and Docker container management together. The VM in `BEE-VM` first need to be build and customized by `BEE-VM Builder` before it can be used. There are two phases in the building process:

1) Phase 1: The first stage in building the VM images for `BEE-VM`. This step is done off-line before it is deployed on targeting HPC system. For uniform standard image building process, we designed our `BEE-VM Image Builder` using Packer [20]. Packer allows us to build customized images automatically. We can also run our own configure scripts in the VM image to enable more flexible customization. Although it is also possible to configure and customize VM OS in phase 2 after booting VMs, the static customization and configuration of images off-line can save a lot of time. For example, building and installing packages. We aim to minimize on-line configuration as much as possible to reduce `BEE-VM` deployment time and put most of the customization into the `BEE-VM Image Builder`. The main customization responsibilities of `BEE-VM Image Builder` include:

1) **Create and configure user accounts:** We need to create a user for the host to login or control the VM.
2) **Configure network interfaces:** Network interface must be configured in advance before `BEE` can configure and control the VM after it has started. However many network configurations cannot be determined until boot time, so we design a customize script built in to the image, so that network interface can be customized automatically when the VM started.
3) **Configure SSH server/client:** SSH is required for `BEE` to control VM and it is also important for MPI. So, we

need to configure SSH key pairs and customized port numbers in this step.
4) **Install essential packages and tools:** Many packages and tools are required including MPI and Docker.
5) **Configure proxy settings:** Most institutional HPC system requires some kind of proxy setting in order to connect to the Internet.
6) **Configure shared storage:** We need to pre-configure the VM image for mounting shared storage system. For example, configuring NFS server.

2) Phase 2: This phase is done when deploying `BEE-VM` on target HPC system. The base image created from phase 1 is highly reusable. Each `BEE-VM` uses the base image to create a dedicated image for the current `BEE-VM`. When `BEE-VM` started, user's Dockerized application is placed in it. This can be done in several ways. User can provide Docker image from public or private DockerHub or build Docker image from Dockerfile. After than, the building process of `BEE-VM` is done.

D. BEE-VM Deployment

Algorithm 2 shows the workflow in `BEE` used for launching `BEE` cluster on a given platform. Although the specific process varies from HPC system to cloud computing system, the general procedures are the same. At first, the launcher needs to know how many and which nodes are available. Then, the user Dockerized application is provided, which can be in either Docker image form or Dockerfile form. Finally, a user-defined hardware configuration file may also be provided, which is used when starting VMs.

There are four stages for launching HPC applications in `BEE`. In the first stage, a new `BEE` cluster is initialized with optional given name. This stage registers each available host to the virtual cluster. Second, `BEE-VM` deploys the VM layer. It creates one VM for each host, and then configures and assigns the VM to the host. In **line 13**, `BEE-VM` starts a VM in parallel using MPI. In the third stage, `BEE-VM` starts to deploy the Docker layer. Depending on what the user provides, `BEE-VM` will either pull the Docker image from public/private registries or build a new Docker image from a Dockerfile loaded into the local VM. Finally, in the forth stage, `BEE` starts the application by launching from the first node (i.e., master node).

IV. BEE-AWS DESIGN

AWS is a highly usable computing environment for cloud and HPC users. To provide a similar Docker-enabled environment on AWS, we designed another back-end `BEE`, the `BEE-AWS`. It enables the user to run their Dockerized application on AWS using `BEE-AWS` the same way as they run on the HPC system using `BEE-VM`. Since both the storage and network on AWS are highly optimized and their configurations are hidden from users, the design of `BEE-AWS` is relatively simpler than `BEE-VM`. In `BEE-AWS`, it first launches an AWS instance based cluster using BOTO API [17] with optimized network configuration and a shared EFS storage. Then, it loads the user application's input data into EFS for storage

6

Algorithm 2 Deploying BEE cluster on HPC/cloud
***
Require: Allocated host nodes: H_1, H_2,..., H_k
Require: User Dockerized application (Docker image/Dock-erfile)
Require: BEE cluster name: cname
Require: User defined hardware configuration: uconf
    // Create a new BEE cluster
1: $BCluster \leftarrow$ create_bee_cluster(cname)
2: **for** $j = 1$ to k **do**
3:     $BCluster$.register_host(H_j)
4: **end for**
    // Deploy VM layer
5: **for** $j = 1$ to k **do**
6:     $vm_j \leftarrow$ create_vm()
7:     vm_j.create_img() // Create image for each VM
8:     vm_j.configure(uconf)
9:     vm_j.setup_shared_vol()
10:    vm_j.setup_network()
11:    H_j.register_vm(vm_j)
12: **end for**
13: $BCluster$.mpi_start_vms()
    // Deploy Docker layer
14: **for** $j = 1$ to k **do**
15:    $dkr_j \leftarrow$ create_doocker()
16:    **if** user provides Docker image **then**
17:       dkr_j.img_pull(d_img)
18:    **else**
19:       dkr_j.img_build(d_file)
20:    **end if** vm_j.register_docker(dkr_j)
21: **end for**
22: $BCluster$.mpi_start_dockers()
23: **[Start user application]**
24: $BCluster.vm_i.dkr_1$.start()
***

sharing. Next, BEE-AWS controls each instance to obtain Docker images either from public/private docker registries or builds Docker images from Dockerfiles. Finally, it starts the user application in Docker containers.

V. BEE-CHAMELEON

Chameleon Cloud [18] is a highly configurable experimental environment for large-scale HPC and cloud research. We also designed a BEE backend, BEE-Chameleon, for delivering same Docker-enabled environment on Chameleon Cloud. So, BEE users can also run their Dockerized application on Chameleon Could without modification. Chameleon Cloud is managed by OpenStack, which allows us to launch, configure, and control instances on Chameleon Cloud through remote APIs. Chameleon Cloud enables user to have root privileges on their bare-metal machines, so we deploy the Docker layer directly on the host without the VM layer in between. Taking advantage of the Infiniband support in SR-IOV enabled instances, we also enable Infiniband support in BEE through a customized Infiniband-enabled BEE Docker base image. So, users of BEE can have an option to utilize accelerated

network in BEE-Chameleon by rebuilding their Dockerized application with our new Infiniband-supported base image.

VI. EVALUATION

A. Storage I/O

Storage I/O is a key component for most scientific HPC applications. For example, large-scale simulations often need to first load large datasets before computation and frequently record the state during computation and once the simulation has concluded. Later, these results may be used for other simulations or for analytics. I/O performance plays an important role for the overall performance of an HPC application. In this section, we conduct experimental evaluation of the storage I/O performance of BEE-VM on an HPC system. In particular, we test and compare the performance of our two storage designs and native performance. To accurately evaluate storage I/O performance, we use the benchmark tool IOR [21]. We simulate a situation where there is one process per node and each process writes and then reads 1GB of a datafile stored in the shared storage directory using MPI-IO functions. To avoid inaccurate results caused by caching, each process only reads files that are produced by another process on a different node.
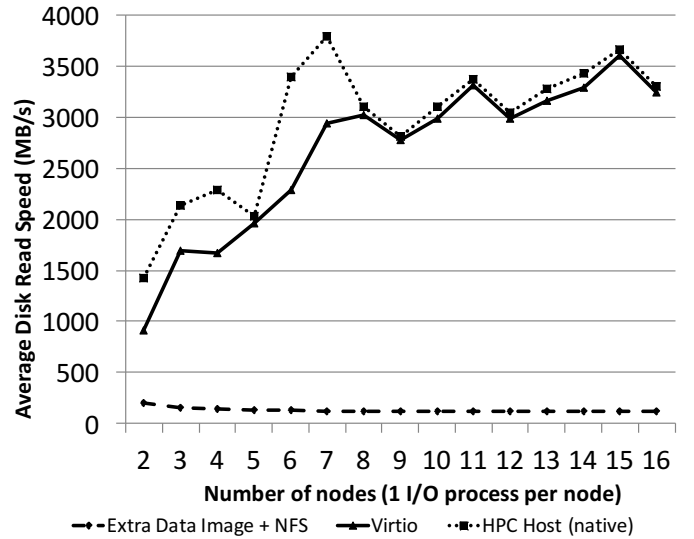

Fig. 6. Storage I/O read test comparison on different storage designs of BEE-VM. Results show that Virtio outperforms Data image + NFS with similar performance to native.

The result is shown in **Fig. 6** for read performance and **Fig. 7** for write performance of our two different storage solutions using different numbers of nodes. Data image + NFS solution does not rely on host file sharing capabilities. When using Data image + NFS design, except the master node which directly mounts the data image though the hypervisor driver, other worker nodes all mount the shared directory through NFS, which depends on the network between master and workers. Since all I/O requests must go through the master node, the network performance of the master node becomes the upper bound of storage I/O performance. As the number of
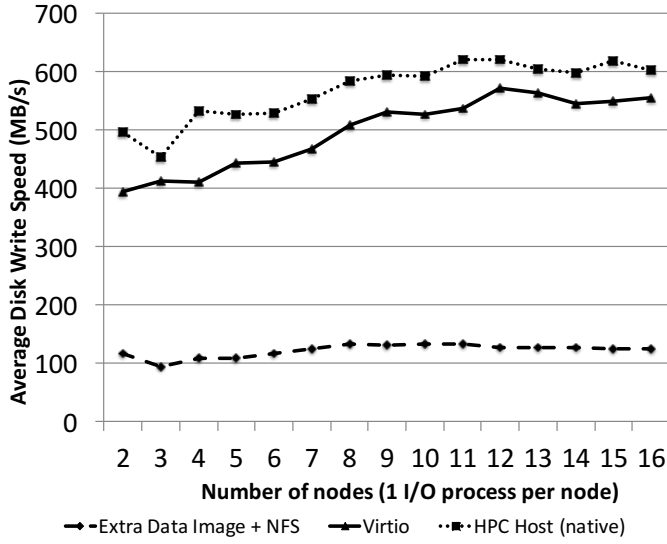
Fig. 7. Storage I/O write test comparison on different storage designs of BEE-VM. Results show that Virtio can achieve 90% of the performance of native.

that the Docker container is slightly better than baremetal in some cases due to the tuning difference under the test scenario.

Fig. 8. IB bandwidth comparison

worker nodes increases, the master node becomes a hot spot, which limits the overall I/O performance. As shown in **Fig. 6** and **7**, the read and write speed of this solution are always near 120-130 MB/s no matter how many nodes participate. `Virtio` design utilizes the file system mapping feature of the hypervisor. All storage I/O requests are sent to and processed by the hypervisor driver. It offers better performance and saves all user space network for MPI communication. As we can see in **Fig. 6** and **7**, this design can achieve almost the same native I/O read performance and 90% native I/O write performance on an HPC system.

B. Network through IB

In this section, we show that network performance result comparison of BEE using the Infiniband (IB) on Chameleon Cloud. The BEE-Chameleon can actually be deployed on any HPC and cloud environment that is equipped with IB and Single Root Input/Output Virtualization (SR-IOV). The target system may or may not have privileged access, so we implemented BEE-Chameleon with two configurations: Docker only (for systems with privileged access) and Docker + KVM (for systems without privileged access). **Fig. 8, 9** show the point-to-point (P2P) network performance. MPI communication function calls between two nodes (one process per node) were used to test the average bandwidth and latency when transferring different message sizes. In **Fig. 10**, we also show the result of network scalability test using MPI all-to-all function call with different message sizes. Limited by the Chameleon Cloud reservation policy, we can only allocate up to 16 nodes. It can be seen that our BEE-Chameleon can provide similar network bandwidth, latency, and scalability compare to the baremetal. It also shows promising trends on larger clusters. Similar results have been seen in [22]. Note
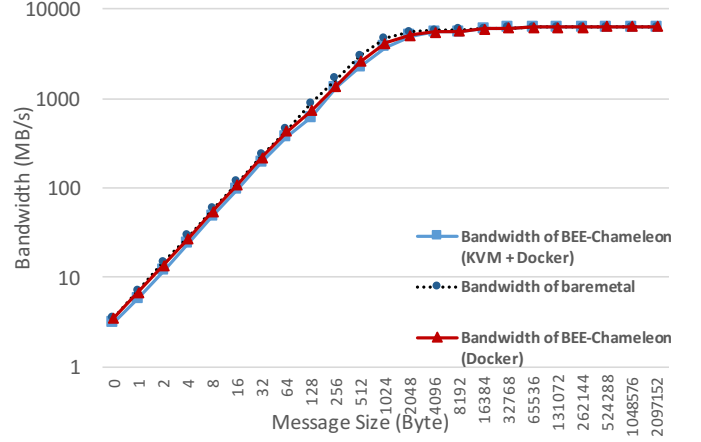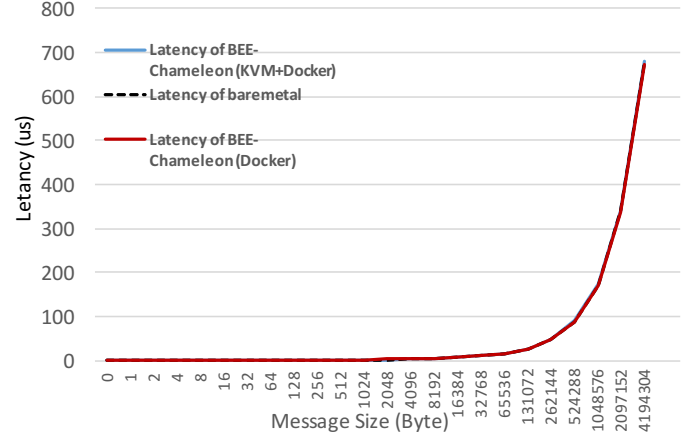

Fig. 9. IB latency comparison

C. Performance Degradation

Computing capability is another factor that needs to be considered for running HPC applications. In this section, we compare the computing performance of an HPC host with KVM-enabled virtual machines and our virtualized `BEE-VM` environment. We also compared the performance on AWS and virtualized `BEE-AWS`. This test is designed to show both the computing power of virtualized CPUs and memory access performance. We run CoMD[23], an HPC proxy application, on a single multicore computing node with KVM enabled.

As shown in **Fig. 11**, although we bring two layers of virtualization for `BEE-VM` and `BEE-AWS`, it only adds slight overhead (approximately 9%) to the application. It also performs well in multicore environments. The overhead percentage almost stays constant for the different number of CPU cores used. This shows that our `BEE` framework using `BEE-VM` and `BEE-AWS` can achieve near native computing capabilities.
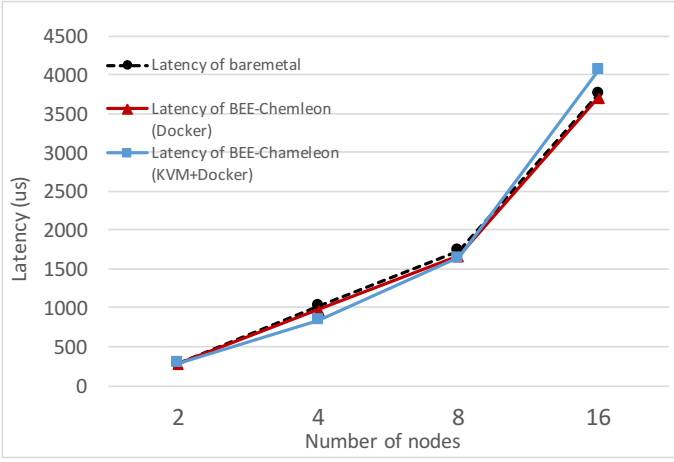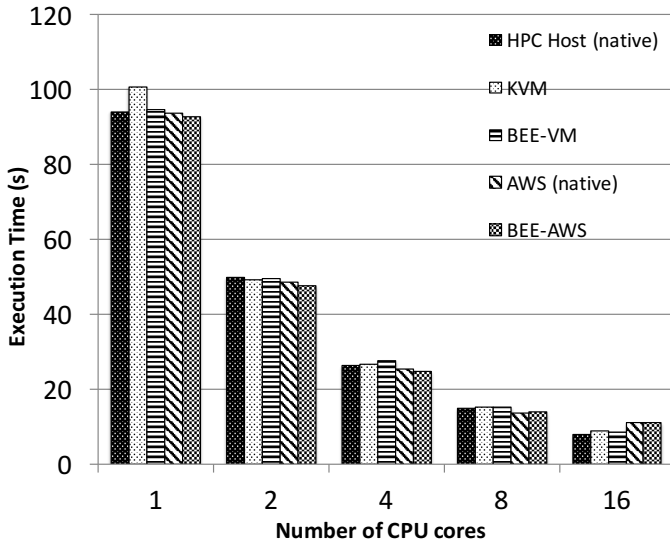
Fig. 10. IB scalability comparison

Fig. 11. Computing performance comparison between HPC host, KVM, BEE-VM, AWS, and BEE-AWS. Both BEE-VM and BEE-AWS virtualized environments provide comparable performance to native performance of HPC and AWS on a single node.

VII. VECTOR PARTICLE-IN-CELL CASE STUDY

In this section, we showcase an example running real HPC applications in `BEE-VM` on a production HPC machine. We test Vector Particle-In-Cell (VPIC) plasma physics code [24], [25], [26] on `BEE-VM`. VPIC is a general purpose particle-in-cell plasma simulation code for modeling kinetic plasmas in multiple spatial dimensions. VPIC is memory bound application that runs on multiple nodes using MPI and pthreads. It has been optimized for modern computing architectures by using short-vector, single-instruction-multiple-data (SIMD) instructions and cache optimization. Before the simulation begins, VPIC needs to load input deck and user configuration files. When computation is finished, VPIC writes the output. With flexible checkpoint-restart semantics, VPIC allows checkpoint files to be read as input for subsequent simulations. Moreover,

VPIC has a native I/O format that interfaces with the high-performance visualization software Ensight and Paraview.

We evaluate the performance of VPIC on our `BEE` framework on an HPC system and a cloud computing system compared with its performance on bare metal HPC systems. For the HPC system, we use our testbed cluster system Darwin. It has a 'Galton' node partition (each 'Galton node' has two NICs), which have KVM enabled. Each node has two 8-core Intel Ivy Bridge E5-2650 v2 CPUs with 251GB RAM. For the cloud system, we use AWS EC2. On AWS, we choose to use `c3.4xlarge` instance type for each node on the cluster we deployed on AWS. Each node is equipped with Intel Xeon E5-2680 v2 CPUs with 16 vCPU cores and 30GB RAM. On Chameleon Cloud, we choose to deploy `BEE` on top of the SR-IOV enabled cluster. Each node in the cluster has two Intel Xeon E5-2670 v3 CPUs, 125GB RAM, and one Mellanox Infiniband card.
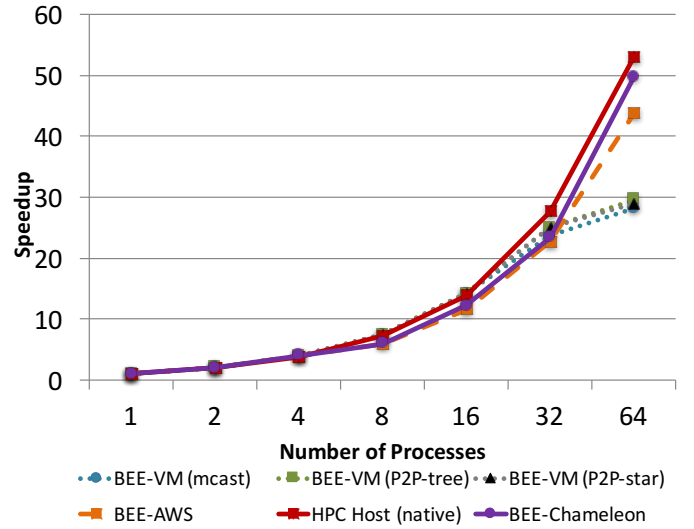

Fig. 12. VPIC scale up test on `BEE-VM`, `BEE-AWS`, `BEE-Chameleon`, and HPC host. Results show that `BEE-AWS` and `BEE-Chameleon` scales similar to HPC native. `BEE-VM` also scales well under 64 processes, but with lower speedup.

We tested VPIC on different environments using 1 to 64 processes. As shown in **Fig. 12**, `BEE-AWS` and `BEE-Chameleon` solution scales similarly to the native host. Our `BEE-VM` solutions (BEE-mcast, BEE-P2P-tree, and BEE-P2P-star) also scale well from 1 to 64 processes. Among all three network configurations, the tree-shaped connection using P2P sockets works the best. Since in VPIC, one-to-one process communication is much more frequent than all-to-all broadcast, tree-shaped connection can greatly mitigate the hot spot problem under this kind of communication pattern, as discussed before. Depending on the communication pattern in different applications, they may work best with different network designs. Due to the overhead of virtualized network, communication intensive applications, such as VPIC, scale up slower on `BEE-VM` than on native HPC hosts beyond 64 processes. We will continue to optimize the network performance

9

(e.g., vNIC configuration, connection styles, topology, etc.).

VIII. Related Work

Containers provide an opportunity to leverage cloud and industry software and practices within the HPC environment. However, various restrictions on current HPC system software and security policies limit the use of industry standard container technologies to execute applications on HPC platforms. Multiple efforts are underway within the DoE laboratories to provide solutions that ameliorate these issues (e.g., Shifter[15], Charliecloud[16], Singularity[27], and BEE-VM) and enable containers to execute securely on HPC platforms.

Shifter is an execution environment that also aims to provide containerized environments for HPC systems. Since deploying the standard Docker daemon on HPC systems imposes security and compatibility issues, they build a Docker-like container environment, which provides portability, isolation, and reproducibility like Docker. Shifter runs customized Shifter images. Docker users need to first import their Docker images and convert them into Shifter images before running. Shifter containers can access the host file system via volume mapping; however, there is no explicit application data management. Also, sharing files between containers requires the file sharing abilities between host machines. Shifter can only be installed on customized Cray machines with root privileges, whereas BEE can run standard Docker images unmodified. Using BEE is easier for developers to ensure consistent environments across their local development and test machines along with production systems. Using standard Docker brings more convenience to develop and distribute Docker images in Docker communities. BEE has explicit application data management that can facilitate easy transfer across host machines for live migration or work flow integration. Also, BEE-VM has several modes for data file sharing between processes that can be configured by the user depending on whether file sharing is enabled on the hosts. If not, BEE-VM can build its own file sharing mechanism, which brings more flexibility. Finally, BEE-VM can be deployed on any HPC system and even cloud systems without root privileges.

Singularity is another containerized execution environment for HPC systems. Similar to Shifter, it also build a Docker-like container execution environment to run customized Singularity images. Standard Docker images are supported but they also need to be converted to Singularity format before running. It is also required to have root privileges in order to install Singularity on HPC systems. Unlike Shifter, Singularity can be deployed on any HPC system. It does not seek to manage application data. Data sharing between containers also depends on the host filesystems. With multiple configuration solutions, BEE has more flexibility for deployment on HPC systems. Besides providing a containerized execution environment, BEE brings better usability by combining data management, workflow integration, live migration, and cloud computing together to provide a more convenient tool for HPC users and developers.

Charliecloud is a container solution that brings Docker-composed environments into HPC systems. It brings many of the benefits of standard Docker container; the main benefit being that users or developers can have consistent building and execution environment from their local development and test machines to large scale production cluster machines. However, it is challenging to install Charliecloud on current HPC systems. Installing Charliecloud environment requires that the target HPC system has a Linux kernel version of at least 3.18. This limitation constrains the current production HPC system ability to support Charliecloud. Moreover, it is only designed as the execution environment, many other aspects of HPC workflow have not been integrated.

While this paper details a specific BEE backend implementation, the BEE-VM and BEE-AWS, there is nothing that precludes the addition of BEE-Shifter, BEE-Singularity, or BEE-Charliecloud backends for specific platforms.

Amazon EC2 Container Service (ECS) [10] is a Docker container service provided by Amazon on AWS. Since ECS deploys on top of EC2, and EC2 has a layer on VMs, ECS actually deploys the Docker container layer on top of the VM layer, providing a similar host-VM-Docker structure as BEE-VM. Regardless of the underlying hardware configuration, ECS provides a consistent building and execution environment by using standard BEE. Docker users can easily run their application on ECS without modification. This allows us to deploy the similar structure on both cloud and HPC environment in BEE.

IX. Conclusions

In this work, we first discussed several workflow and productivity problems that current HPC users are facing. Then, we analyzed the potential of integrating commonly used Docker container technology into the existing HPC systems to build a consistent container environment for HPC users. Following this goal, we proposed several requirements that a successful containerized HPC framework should achieve and we designed a Docker container-enabled HPC framework, BEE, and a universal backend for HPC systems, BEE-VM. BEE satisfies all the requirements we set and also allows users to use the compute resources of both existing HPC systems and cloud computing systems with little extra configuration. Experiments show that BEE-VM can achieve comparable storage I/O performance and computing capability. Finally, a case study on a widely used VPIC simulation tool was tested on BEE and native HPC systems for performance comparison.

References

[1] G. Vallee, T. Naughton, C. Engelmann, H. Ong, and S. L. Scott, "System-level virtualization for high performance computing," in *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, 2008.

[2] A. Reuther, P. Michaleas, A. Prout, and J. Kepner, "Hpc-vms: Virtual machines in high performance computing systems," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, 2012.

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

[4] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance vmm-bypass i/o in virtual machines." in *USENIX Annual Technical Conference, General Track*.

[5] J. Zhang, X. Lu, S. Chakraborty, and D. K. D. Panda, "Slurm-v: Extending slurm for building efficient hpc cloud with sr-iov and ivshmem," in *European Conference on Parallel Processing*, 2016.

[6] S. Gugnani, X. Lu, and D. K. Panda, "Performance characterization of hadoop workloads on sr-iov-enabled virtualized infiniband clusters," in *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, 2016.

[7] A. Tikotekar, G. Vallée, T. Naughton, H. Ong, C. Engelmann, and S. L. Scott, "An analysis of hpc benchmarks in virtual machine environments," in *European Conference on Parallel Processing*, 2008.

[8] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda, "Virtual machine aware communication libraries for high performance computing," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

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

[10] "Aws container service," URL : https://aws.amazon.com/ecs/.

[11] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, 2015.

[12] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, 2014.

[13] B. Ruan, H. Huang, S. Wu, and H. Jin, "A performance study of containers in cloud environment," in *Advances in Services Computing: 10th Asia-Pacific Services Computing Conference, APSCC 2016, Zhangjiajie, China, November 16-18, 2016, Proceedings 10*, 2016.

[14] A. S. Harji, P. A. Buhr, and T. Brecht, "Our troubles with linux kernel upgrades and why you should care," *ACM SIGOPS Operating Systems Review*, 2013.

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

[16] G. M. Kurtzer, "Singularity 2.1.2 - Linux application and environment containers for science," 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60736

[17] "Boto api," URL: https://github.com/boto/boto3.

[18] "Chameleon cloud," URL : https://www.chameleoncloud.org/.

[19] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, 2008.

[20] "Packer," URL : https://www.packer.io/.

[21] "Ior benchmark," URL: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/ior/.

[22] J. Zhang, X. Lu, and D. K. Panda, "Performance characterization of hypervisor-and container-based virtualization for hpc on sr-iov enabled infiniband clusters," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1777–1784.

[23] "Comd," URL:https://github.com/exmatex/CoMD/.

[24] 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*, 2008.

[25] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultra-high performance three-dimensional electromagnetic relativistic kinetic plasma simulation a," *Physics of Plasmas*, 2008.

[26] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, "Advances in petascale kinetic plasma simulation with vpic and roadrunner," in *Journal of Physics: Conference Series*, 2009.

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