Build and Execution Environments (BEE) : an Encapsulated Environment Enabling the HPC Application Running Everywhere

Anonymous

ABSTRACT

Because different HPC system have different software and hardware configurations, configuring and building before running HPC application is necessary. However, it can take a lot of time and effort for application users and requires application users to have enough application technical knowledge. Also, when users want to run legacy application developed years or even decades ago, required dependency libraries maybe hard to find or have difficulties compatible with current software stack. So, having a consistent execution environment across HPC system in different locations or time is desirable, however it is usually hard to accomplish on bare metal hosts. Container technology like Docker brings great benefits to application's development, build and deployment process. While most cloud computing systems are already equipped with Docker, not much work has been done to deploy Docker on HPC systems. In this work, we propose a Docker-enable build and execution environment for HPC system – `BEE`. It brings all the benefits of Docker to existing HPC machines without installing software and with no system administrator configuration. We show that current HPC application can be easily configured to run BEE. It eliminates the need for reconfigure and rebuild applications for different systems and it also provides comparable performance.

KEYWORDS

High Performance Computing, Cloud Computing, Container.

ACM Reference format:

Anonymous . 2017. Build and Execution Environments (BEE) : an Encapsulated Environment Enabling the HPC Application Running Everywhere. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

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

Because different HPC system usually have different software and hardware configurations, HPC users must configure and build their application for each specific machine, which has tedious and time consuming and has become a bottleneck to productivity. Moreover, in some cases, legacy applications still serve as key components in the process of scientific or industry research. However, legacy applications are developed years or even decades ago. There is no support nowadays. Those applications may require specific old versions of dependency libraries and certain kinds of hardware in order to make them run normally and output desired results. However, it maybe hard to find libraries that meet the requirement of legacy applications and compatible with current HPC systems. Even worse, legacy applications may not even designed and built for modern computer architectures. These requirements impose great challenges for HPC users. Also, on shared resource computing environment, users must implement checkpoint/restoration to stop and restore their computations across allocations, because of time or resource limitations. These checkpoints cannot be migrated easily across different systems that have different software and hardware configurations. A consistent execution environment is also important for HPC application developers. The consistent between developing, testing, and production environment can greatly save developers' time on fixing compatiability issues, which can significantly accelerate developing process.

Virtualized environment, especially virtual machine, has been well studied in HPC system to provide more consistent, isolated and secure environment [19, 22]. For example, [12] built a virtualized HPC environment using Xen-based virtual machines. By leveraging high performance I/O passthrough [15], it can achieve near-native performance when running HPC benchmarks. In [23] showed at current resource manager in HPC system cannot well supervise virtual machines and associated critical resource, so they proposed a Slurm-V, which extends Slurm with virtualization-oriented capabilities. [10, 21] characterized the performance of running multiple kinds of application on virtualized HPC clusters. [] 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 a MPI library with IVC enabled.

However, managing virtual machine image is not trivial, since virtual machine image need to contain files of operating system, dependent libraries/packages, user applications and input/output data, which may takes several gigabytes on disk space. Migrating images between HPC systems or distribute images among computing nodes can takes a lot of time. On the other hand, Linux container technology like Docker [2] [1] enables consistent execution environment for development, build and deployment. By developing using Docker, developers only need to build their application once in Docker on their local machine, then the application can run on any Docker-enabled machine. It only needs to ship the Docker images to the target machine. Docker image only contains dependent

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

libraries/packages and user application, which has much less size than virtual machine images [5]. Also, since Docker creates a thin translation layer that allows the guest application to share the host operating system, the performance penalty is negligible [17].

It would be of great benefit to bring the advantages of Docker realized in the cloud to HPC users, however, Docker is not usually installed on current HPC systems. The main reason behind it is that Docker requires Linux kernel version to be higher than 3.8, but current main stream Linux kernel version is far behind this version [11]. For compatibility and security consideration, it would take years before HPC systems can upgrade their Linux kernel.

HPC systems that run containers are being actively deployed using software such as Shifter [13] and Singularlity [14]. BEE's approach is complementary. HPC systems that have been built and deployed with Shifter or Singuarity can run containerized applications without a virtual machine layer. Applications must be built for Singularity or Shifter. BEE brings containerized applications to all other HPC machines and, because it runs Docker natively, does not require a rebuild when moving applications across machines.

A successful Docker-enabled HPC workflow framework should provide:

(1) **Automation:** Both the deployment of container-enabled execution environment and user's application should be done automatically. Configuration, building and management should be hidden from users, so that users don't need extra expertise and technical knowledge to run their application.
(2) **Portability:** The execution framework should run across multiple platforms, including HPC system and cloud computing system.
(3) **Flexibility:** Users should have the flexibility choose different platform as their needs change and also migrate running applications to different platform. The framework should also allow users to build their workflow logic into the runtime environment.
(4) **Reproducibility:** Applications should behave exactly the same across platforms regardless the underlying software and hardware configurations.
(5) **Interfacing Continuous Integration:** Continuous Integration(IC) has been widely used in HPC application development workflow. The execution framework should be interfacing IC, so that it can effective free HPC domain users from handling complicated application configurations.

In this work, we propose a new build and execution environment that can enable Docker on almost any current HPC systems. We call it Build Execution Environment (BEE). To overcome the Linux kernel version issue and provide more flexible design environment, the BEE system configures a virtual machine (BEE-VM) for each host and deploys docker in this VM. For machines that support KVM (on by default in Linux), a hardware accelerated type 1 KVM hypervisor provide bare metal performance. For machines without KVM, we build and configure QEMU in user space. QEMU runs on many host operating systems and on all Linux since kernel 2.6, which makes BEE compatible with almost all HPC machines

The most important, our solution does not require root/administration privileges of the HPC system, so any user can deploy BEE on HPC systems. Because we do not customize Docker, users can always get benefits from the features of the latest Docker. The contributions of this work include:

(1) **Docker-enabled environment on HPC system:** By using BEE, we can provide Docker-enable environment into current HPC systems. HPC users can easily Dockerize their application and 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) **Work flow integration:** The containerized environment provided by BEE can greatly facilitate the HPC work flow integration without complicated configurations. Different parts in the work flow can be packed in container and viewed as modules, so we can easily integrate different modules together to form the work flow we want using BEE.
(3) **User space deployment on unmodified HPC systems:** On all QEMU compatible systems, users can deploy BEE on the HPC system they are using without root account.
(4) **Standard latest Docker support:** We run unmodified Docker daemon inside BEE, so users can Dockerize their application in the standard way. Existing Dockerized application can run on BEE unmodified. Also, Docker inside BEE can be upgraded so that users can always benefit from the latest version of Docker.
(5) **Interfacing Continuous Integration:** Many Continuous Integration (CI) of current HPC application development project are using Docker as their output. Since our BEE framework support standard Docker, users can run CI-generated Dockerized application on BEE without any manual configuration.
(6) **Both software and hardware virtualization:** Existing solution run container direct on the host, however different hardware on different host may yield different result due to different hardware architect. Our solution, on the other hand, provided another hardware virtualization layer, so that together with Docker container we provided both hardware and software consistent environment. This further enhanced the reproducibility of BEE.
(7) **Hybrid HPC and Cloud environment:** Besides HPC systems, BEE can also be deployed on cloud computing environment, such as Amazon Web Services(AWS). We provide the same standard Docker environment for users, so that users can run their HPC application on the cloud without modification.

The rest of this paper is organized as follow: In section 2, we discuss the design details of BEE with evaluation in section 3. In section 4, we provide case study on real HPC application running on BEE. In section 5, we discuss other large scale container-enabled execution environment and how BEE it different from them. Finally, we discuss future work in section 6 and make conclusion in section 7.

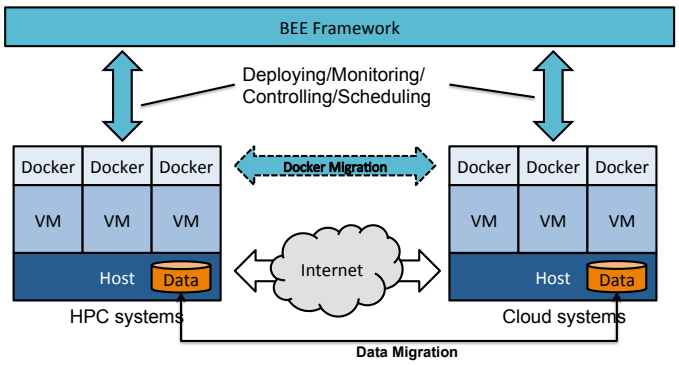
2 DESIGN

2.1 Overall Design

Linux container technology like Docker enables consistent software and hardware environment for development, build and deployment. By developing using Docker, developers only need to build their application once in Docker on their local machine, then the application can run on any Docker-enabled machine. However, Docker is not usually installed on current HPC systems. The main reason behind it is that Docker requires Linux kernel version to be higher than 3.8, but current main stream HPC system have lower kernel version. For compatibility and security consideration, it would take years before HPC systems can upgrade their Linux kernel. Existing solution on current HPC systems either requires specially customized HPC system or Docker daemon. However, those customizations would limit the practicability to deploying such system to other machines. First, most HPC systems are not allowed to be customized either in software or hardware by normal users. Second, customized Docker daemon could bring compatibility and security issues.

To overcome the Linux kernel version issue and provide more flexible design environment, we first create a extra virtual machine layer on top of the host and then deploy Docker on the virtual machine layer as shown in **figure 1**. Besides brings standard latest Docker to HPC system, the addition virtual machine layer also brings following advantages: (1) Since we have full control in the virtual machine, we can create consistent environment for upper Docker layer across platforms. For instance, we can also deploy BEE on both HPC and commercial cloud computing system (e.g. AWS), which allow Dockerized HPC applications to run on unmodified. (2) Docker only brings virtulization on software/OS level. The additional virtual machine layer brings extra hardware level virtulization, which further ensure long-term reproducibility. (3) Docker level live migration requests hosts(in prospective of Docker container) OS version to be similar between checkpointing side and restoration side. This extra virtual machine can guarantee that requirement is met. This can be a problem if Docker is run directly on the hosts.

Figure 1: BEE Framework



2.2 Network Design

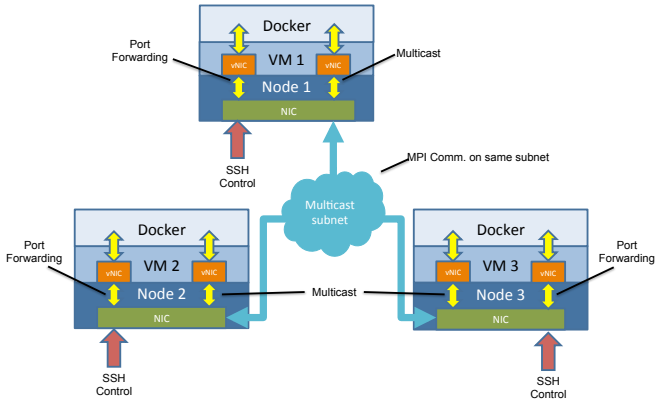
In this section, we discuss our network design in BEE. Network part of BEE is mainly used for two functions. The first one is that we need to dynamically configure and do deployment in virtual machine and Docker container. This need to be down automatically and remotely. Also, since we are aiming at HPC application and most HPC application requires MPI, so that we also need to build network for MPI communication between different computing nodes. For the controlling virtual machine and Docker, we choose to use Secure Shell(SSH). It has several advantages: (1) SSH server and client can be easily deployed on most machines and it is usually enable on HPC systems; (2) RSA key pairs can be used to for simple and secure machine control without worrying about complicated password; (3) SSH allows remote command execution without login, which can facilitate batched virtual machine/Docker control. Basically, we need to run SSH server inside virtual machine and Docker container. To avoid port conflicting, we configure the SSH server to listen to different ones than the host, since HPC won't allow normal user to configure SSH port. As for the MPI communication, we need to create a virtual subnet comprises all virtual machine or Docker container, so that they can communicate.

2.2.1 Virtual machine layer. First, we design the network between virtual machine. This is necessary since the Docker layer on top of it would depends on the network here. The main challenges of designing network for virtual machine is configure virtual network card without root privileges, since that is the most common case for HPC users. We designed two solutions for different hardware configurations on the HPC systems.

To connect virtual machine to the host network, many commonly used approaches requires root privilege. However, in this work, we restrict our scope to only use normal user level privilege, so we do not consider technologies like TAP in our design. In order to enable SSH connection to the virtual machine through host, hypervisor much be configured to use port forwarding to map an unused port on host to the SSH port on the virtual machine. However, this makes the network interface card(NIC) unusable for MPI. Although some MPI libraries can be configured to use designated port and can cooperate with port forwarding, many commonly used MPI libraries (e.g. OpenMPI) uses random port, which is not easy to work with port forwarding mechanism. So, we build a second virtual NIC dedicated for MPI communication. We design two solutions to cooperate virtual NICs with physical NICs on the host.

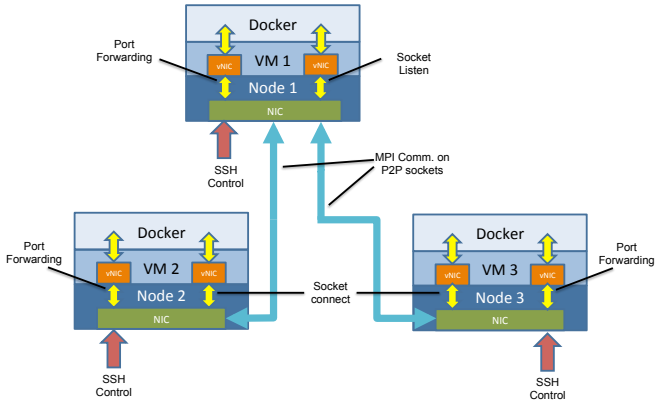
Solution 2: 1 NIC + multi-cast To accommodate more general case, in which each computing node has one physical NIC, we design our second network solution. In this solution, we still use port forwarding to combine the SSH virtual NIC with the physical NIC. For the second virtual NIC for MPI, we use multi-cast subnet to connect all second virtual NICs of all virtual machines together. Since all virtual NICs are connected to the same subnet, there is no restriction on port usage, so any MPI library can be used. The advantages of this design is that the configuration is simple and if one node in the multi-cast network failed, it won't affect the connect in between other nodes, which can cooperate with simple fault tolerance mechanism. However, multi-cast network also brings higher communication overhead. Although MPI broadcast communication behaves similar in the multi-cast subset, point-to-point MPI communications are converted to multi-cast, which generated more data packages in the subnet.

Figure 2: Network Design using one NIC and multicast



Solution 3: 1 NIC + P2P sockets To keep using one physical NIC and reducing network communication overhead, we design our third solution. Different than our second solution, instead of connecting the second virtual NIC to a multi-cast subnet, we connect each second virtual NIC from each virtual machine using point-to-point socket connection. This is similar to the wired Ethernet connection between computing node in HPC system. Since there is point-to-point routing path between nodes, there are no extra unnecessary data packages in the network. However, socket connects between nodes may route via inter-median nodes, so if one node fails, it may breaks several connects multiple node, so more complication fault tolerance mechanism need to be adopted. Also, the performance of this kind of network is affected by the connection pattern between nodes, so we adopt two connection pattern in this solution.

Figure 3: Network Design using one NIC and P2P sockets



(1) **Star-shape connection** We first connect the nodes in start-shape with master node in the center. This connection pattern can effectively limit connection hops between each pair of nodes. However, since all communications must go through the center node, it may become a hot-stop in the network especially for application involves intensive network communication or the shared storage design needs network communication(will be discussed in later section).

(2) **Tree-shape connection** Then we connect nodes in tree-shape which master node as the root. especially, we use binary tree structure. This connection can mitigate hot-stop issue, but it also more connection hops between each pair of nodes.

2.2.2 Docker container layer. Based on the network of virtual machine layer, we now design the network between Docker container. There are several network configuration for Docker container. To minimize network overhead, the most direct way is network pass-through. Basically, all network interfaces on the host (i.e virtual machine in our case) will be exposed to Docker container. Since there is not additional buffer or translation in between, it usually bring the lowest overhead. However, it is also considered to be the least unsecured since everything is exposed. But we build Docker inside virtual machine, this extra layer already provide enough isolation, so there is no extra security issue there. So, we choose pass-through as the network mode used for Docker container(i.e. host mode). Since network interface is exactly the same between virtual machine and the Docker container inside it, the software level configuration (e.g. SSH, MPI) are similar and easy to configure.

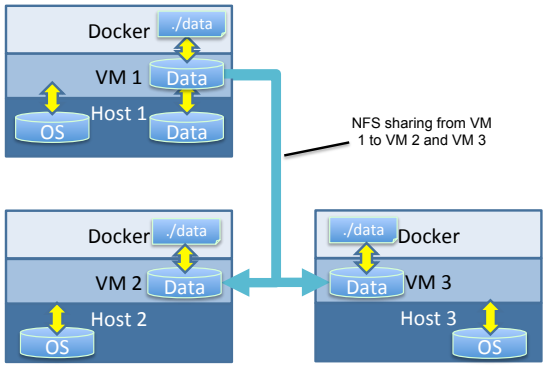
2.3 Storage Design

In this section, we design the share storage system of BEE. General HPC application usually use some kind of shared file system(e.g. NFS) to share data between processes. To provide the same environment, we need to build a shared file system across different nodes in BEE. To between integrate user's work-flow logic, we aim to separate data from run-time data and operation system itself. This separation allows users to easily migrate their data to other platform without extra space overhead. We design two architectures to allow different processes in different nodes to share files in real-time.

2.3.1 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, filed updated by one machine is visible by other machine in real-time. So, we choose to mount this data image only to the master node, and use NFS to share mounted disk with other virtual machines. By using the extra data image, data can be easily migrated. The input data is first pre-loaded in the image, which can result from other part of user's work flow. After execution, the output data is also stored in this image. To move the data to other part of the work flow, user only need unmount the data image from current computing node and mount to the computing node used for the next stage. Also, the image encapsulation can protect data integrates. However, the data image is shared with other worker nodes via NFS, which highly depends on the performance of virtual network. If user's application is I/O extensive and the network bandwidth is limited, it may consume too much network resource, which may degrade MPI communication performance and slows down users application.

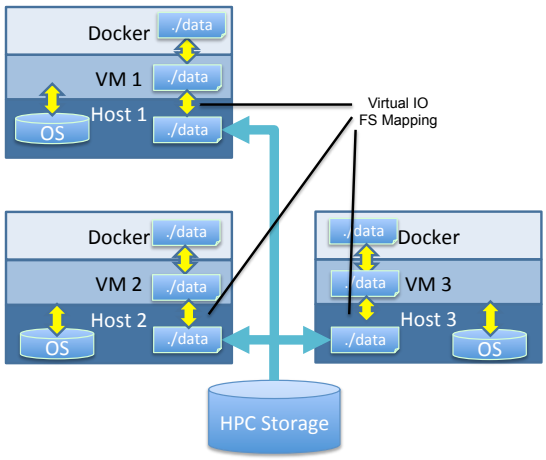
2.3.2 Virtual IO. In our third solution, we eliminated all the dependency on virtual network. We use the virtual IO [20] feature in QEMU to map a host directory to a directory in virtual machines. It only required minimum configuration at virtual machine boot time. Similar to solution 2, each machine are mapping the same directory, so the data is still shared using NFS system in HPC system and it is

Figure 4: Shared Storage Design using Extra Data Image + NFS



visible to the host in real-time. Since it does not relay on network, the whole virtual network is saved for MPI.

Figure 5: Shared Storage Design using Virtual IO



2.3.3 Data sharing between Docker containers. Finally, as for data sharing in Docker layer, we use the data volume sharing feature in Docker to mount the shared folder in virtual machine to a directory in Docker. Since Docker runs as process in virtual machines, sharing data in between brings negligible overhead. This configuration is also compatible with all data shared mechanism in virtual machine layer.

2.4 BEE-VM Image Builder

Besides network and shared storage design, designing virtual machine image also plays an important role in BEE. Virtual machine is the core of BEE that combines computing resources from host, virtual network, shared storage, user provided Dockerized application and Docker container management together. Customizing virtual machine image is the key for BEE. For uniform standard image building process, we design our BEE-VM Image Builder using Packer [4]. Packer allows us to build customized image automatically. We can also run our own configure scrips in the virtual machine image to enable more flexible customization. Although it is also possible to configure and customize virtual machine OS after booting virtual machines, customize and configure images off-line can save a lot of time. For example, building and installing packages. We aid to minimize on-line configuration as much as possible to reduce BEE deploying time and leave most of the work to BEE-VM Image Builder. The main customization responsibility of BEE Image Builder include:

(1) **Create and configure user:** We need to create a user for the host to login or control the virtual machine.
(2) **Configure network interfaces:** Network interface must be configured in advance before BEE can configure and control the virtual machine after it has started. However many network configuration 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 virtual machine started.
(3) **Configure SSH server/client:** SSH is required for BEE to control virtual machine and it is also important for MPI. So, we need to configure SSH key pairs and customized port number 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 virtual machine image for mounting shared storage system. For example, configuring NFS server.

2.5 Work Flow Integration

By encapsulating each components of HPC or cloud computing system into container, each component can be treated as modules that can be easily assembled into a large system as needs. This enables user to easily build their work flow logic into BEE. Once user has set the work flow logic, BEE will automatically deploy and manage users application according to the logic. For example, as show in **Figure 6**, when user wants to a series of pipelined simulations, the user need to manually configure and start each simulation one by one and also need to handle data transfer for the pipeline logic. If the pipeline consists many simulations, it can be considerable hard to manually deploy and debug, especially when different simulations are conduced on different hosts. However, by using BEE, user only need to indicate which simulations need to run and which one follow the other, then BEE can automatically deploy simulation applications in order and setup the pipeline data transfer. The containerized environment also allows user to change certain part work flow logic at runtime as long as it does not interrupt normal execution. For example, if during the pipelined simulation, some bug occurs and causes incorrect final output data. In-situ analysis is a common approach to diagnose the problem by checking intermediate result between two simulation. User need to login to each hosts and manually deploy some in-situ analysis tools, which can be hard when the original work flow logic is complication. By using BEE, the in-situ analysis tool can be encapsulated as another user application and is able to be deployed into users work flow at real-time.

Continues Integration(CI) [9] has been widely used in many HPC application development process. It used for unit testing, fixing compatibility issue during integration, etc. Many development project choose to use standard Docker image as output application format. Since BEE takes standard Docker image as input, it can seamlessly interfacing CI development.

2.6 Hybrid HPC and Cloud BEE Framework

In this section, we discuss the overview of our BEE framework. Besides HPC computing resources, we also aim to utilizes computing resources from cloud environment. Efficiently manage available resources and deploying virtualized execution environment is one of the keys for providing high performance virtualized environment [12, 16], so we design the BEE framework to efficiently manage our multi-layer virtualized environment. As shown in **Algorithm 1**, we outline the general workflow of BEE. First, user need to provide all the computing resources available to that user. This can be both HPC system and cloud computing system. User need to indicate which nodes are available and current allocated time slot. User also need to indicate the priority of using these computing system. For example, when running a compute intensive application, user may want prioritize system with high performance to the front of the list. Then user need to provide their Dockerized application with input data and hardware configuration file indicating on what kind of system they want their application to run.

Before execution, BEE picks the system from the front of the list(**Line 3**), which has the highest priority. Then it check if current application is in initial stage or needs to restore from previous checkpoint. If it is in initial stage, BEE just attach the input and run the application as shown in **Line 10 and 13**. Otherwise, BEE first migrate previous intermediate data and checkpoint from last system to current one (**Line 5 and 6**) then attach the data and restore Docker checkpoint (**Line 7 and 8**)before running.

During execution, BEE monitor the current state of the running BEE cluster. When it is closing to the end of current time slot, BEE checks if current application has completed its execution. If so, BEE stops the cluster and output the result. Otherwise, BEE initiate checkpointing procedure. It first pause current execution to take a checkpoint of each Docker container and detached the intermediate data. It marks *need_migration* has true to ensure the current application will resume from current execution stage next time. When finish checkpointing procedure, BEE will check for next available system to run. If no other computing system available, BEE will store current data and checkpoint data in disk, until user indicate new computing resources are available.

Algorithm 2 shows the work flow 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 procedure are the same. At first, the launcher needs to know how many and which nodes are available now. Then the user Dockerized application is provided, which can be in either Docker image form or Dockerfile form. finally, user defined hardware configuration file is also provided, which is use when starting virtual machines.

There are four stage for launching application in BEE. In the first stage, a new BEE cluster is initialized with optional given name. This stage basically register each available host to the virtual cluster.

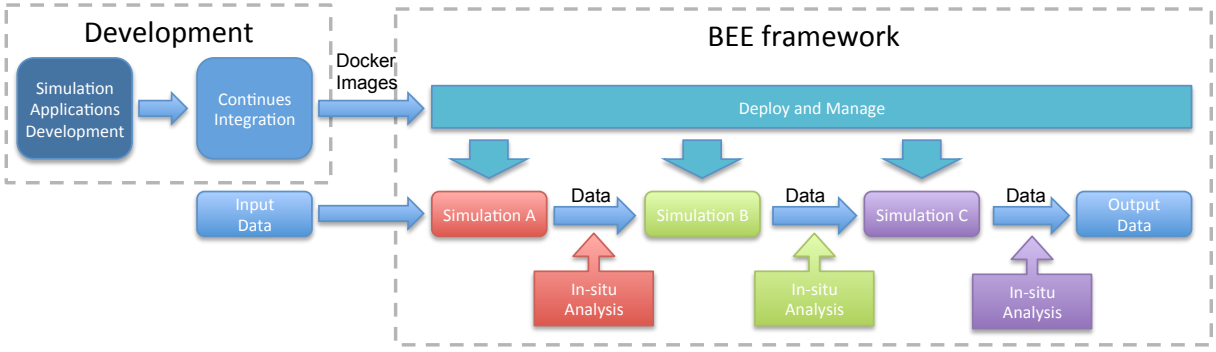
Algorithm 1 BEE Framework

Require: $HPC/Cloud_1$: [Host nodes: H_1, H_2,..., H_k][Time slot: T_1]
Require: $HPC/Cloud_2$: [Host nodes: H_1, H_2,..., H_k][Time slot: T_2]
...
Require: $HPC/Cloud_m$: [Host nodes: H_1, H_2,..., H_k][Time slot: T_3]
Require: Computing resource priority list: L
Require: User Dockerized application(Docker image/Dockerfile)
Require: Input data: D
Require: User defined hardware configuration: uconf

```
1:  need_migration ← False
2:  lastHost ← N/A
3:  while H ← get_top(L) do
4:      if need_migration is True then
5:          data_tranfer(lastHost, H, D)
6:          checkpoint_tranfer(lastHost, H, DCHK)
7:          attach_data(D, H) // Load data into H
8:          mpi_docker_restore(H, DCHK)
9:      else
10:         attach_data(D, H)
11:     end if
12:     deploy_bee(H, user_application, uconf)
13:     while H running normal AND Running-time not close to T
        do
14:         Monitor(H)
15:     end while
16:     if execution incomplete then
17:         pause(H)
18:         DCHK ← mpi_docker_checkpoint(H)
19:         D ← detach_data(H) // Unload data from H
20:         need_migration ← True
21:         lastHost ← H
22:         stop(H)
23:     else
24:         stop(H)
25:         result ← D
26:         terminate()
27:     end if
28: end while
29: if execution incomplete then
30:     DCHK ← mpi_docker_checkpoint(H)
31:     D ← detach_data(H)
32:     stop(H)
33:     store D and DCHK when no resource available
34: end if
```

Second, BEE deploys the virtual machine layer. It creates one virtual machine for each host, then do configuration and assign virtual machine to host. In **line 15**, BEE start virtual machine in parallel using MPI. In the third stage, BEE starts to deploy the Docker layer. Depending on what user provide, BEE will either pull Docker image from DockerHub or private registry or build Docker image from Dockerfile in local virtual machine. Finally, in the forth stage, BEE starts the application by launching from the first node (i.e. master node).

Figure 6: BEE with Workflow Integration



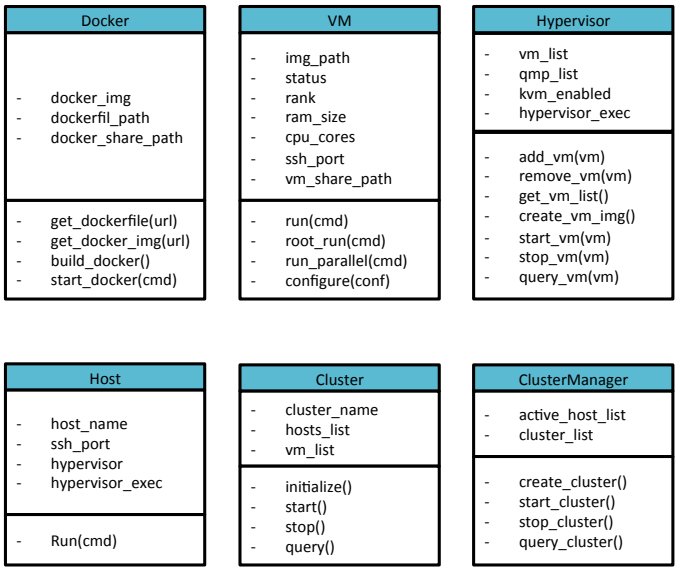
Algorithm 2 Deploying BEE cluster on HPC/cloud system

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

2.7 BEE Framework Design

In this section, we discuss the design details of our BEE framework. To better manage large scale BEE clusters and for extensibility consideration, we choose to use object-oriented design in python for BEE as shown in **Figure 7**. We design several key classes for key components in BEE, including: `Docker`, `VM`, `Hypervisor`, `Host`, `Cluster`, and `ClusterManager`. `Docker` class stores

Figure 7: BEE Object-Oriented Design



Docker-related information. For example, Docker image name, Dockerfile path, Docker's running state and directory in Docker that is shared between Dockers. It also contains necessary functions to control or run command in the Docker containers. `Docker` class stores virtual machine-related information, including the Docker class instant, current hardware configurations, network settings, virtual machine image file path, shared storage directory, etc. It also has necessary functions to control the virtual machine. However, we leave start/stop/query function to the `Hypervisor` class, since implementation of those functions are hypervisor-specific. `Hypervisor` class stores hypervisor-related information. For example, the path to the hypervisor binary, KVM availability, and a list of virtual machines that it is currently running. Since we allow different machine to use different hypervisors, we inherent `Hypervisor` class to get hypervisor class for specific hypervisor. For example, we have a `QEMU` class that is targeting for QEMU hypervisor. Besides all the functions inherent from `Hypervisor` class, it also has some

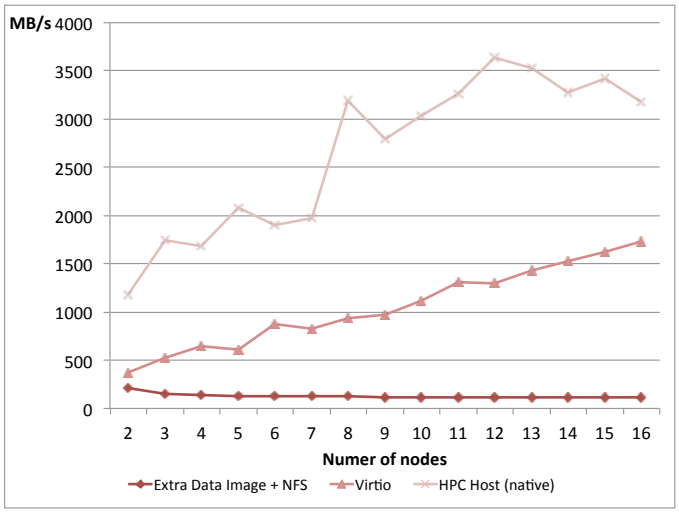
QEMU-specific function, like QMP virtual machine monitor functions. `Host` class maintains all the information of each host, including the host name, username, port number and the hypervisor that is currently running on it. It also has functions used to execute command on the host. `Cluster` class maintains the information of a virtual BEE cluster, including cluster name, all host nodes and virtual machines involves, network configuration for this cluster, etc. It also has functions to control and query the cluster. `ClusterManager` class is used to manage multiple clusters on a computing system. It maintains all the active host nodes on the current system, so that they can be shared between cluster. It provide necessary functions to control each clusters.

3 EVALUATION

3.1 Storage I/O test

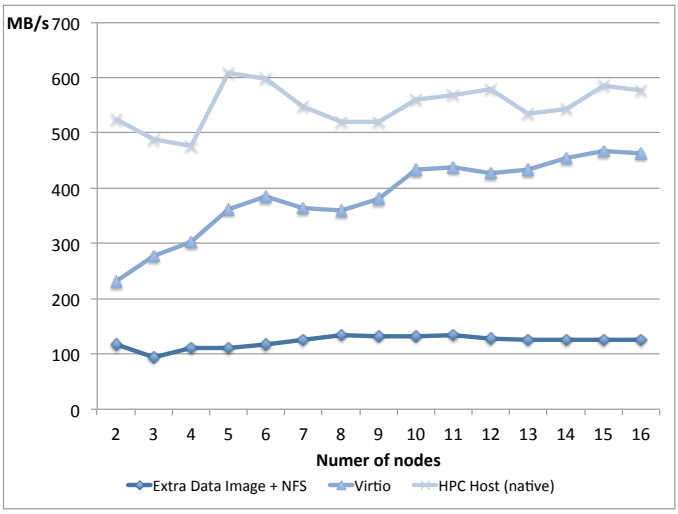
Storage I/O is a key component for most scientific HPC applications. For example, large scale simulations need to first load large amount of data before processing and write back result after. Later, the result maybe used for other simulations. So I/O performance plays an important role for the overall performance of HPC application. In this section, we conduct experimental evaluation on storage I/O performance of BEE on HPC system. specially, we test and compare the performance of our three storage designs and native performance. To accurately evaluate storage I/O performance, we use benchmark tool IOR [3]. We simulation the situation where there is one process per node and each process try to write and read 1GB of datafile stored in the shared storage directory using MPI-IO functions. To avoid inaccurate result cause by caching, each process only read files that is produced by other process on different node.

Figure 8: Storage I/O read test comparison on different storage designs of BEE



The result is shown in **Figure 8** for read performance and **Figure 9** for write performance of our two different storage solution using different number 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 mount the data image though hypervisor driver, other worker nodes all mount the shared directory through NFS, which depends on the network between master and worker. Since all I/O requests must go through the master node. The network performance of master node becomes the upper bound of storage I/O performance. As the number of worker nodes increases, master node becomes a hot spot which limits the overall I/O performance. As shown in **Figure 8** and **9**, the read and write speed of this solution is always near 120-130 MB/s no matter how many nodes is participated. `Virtual IO` design utilizes the file system mapping feature of hypervisor. All storage I/O requests are sent to and processed by hypervisor driver. It offers the better performance and saves all network for MPI communication. As we can see in **Figure 8** and **9**, this design can achieve almost half of the I/O read performance and 80% I/O write performance of bare metal HPC system. The performance continues to improve as we increase the number of nodes.

Figure 9: Storage I/O write test comparison on different storage designs of BEE



3.2 Computing capability test

Computing capability is another factor that need to be considered for running HPC applications. In this section, we compare the computing performance of HPC host with our virtualized BEE environment. This test is designed to show both the computing power of virtualized CPUs and memory access performance. We choose to run a compute intensive application, LU factorization, on a single multicore computing node with KVM enabled.

As shown in **Figure 10**, although we brings two layers of virtualization for BEE, it only bring slightly overhead (approximately 9%) to the application. It also performs well in multicore environment. The overhead percentage almost stays constant for different number of CPU cores used. This shows that our BEE framework can achieve near native computing capabilities.

Figure 10: Computing performance comparison between HPC host and BEE

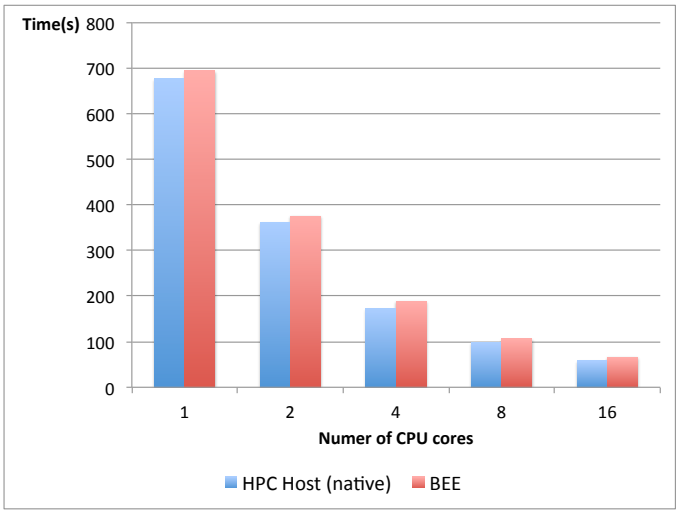
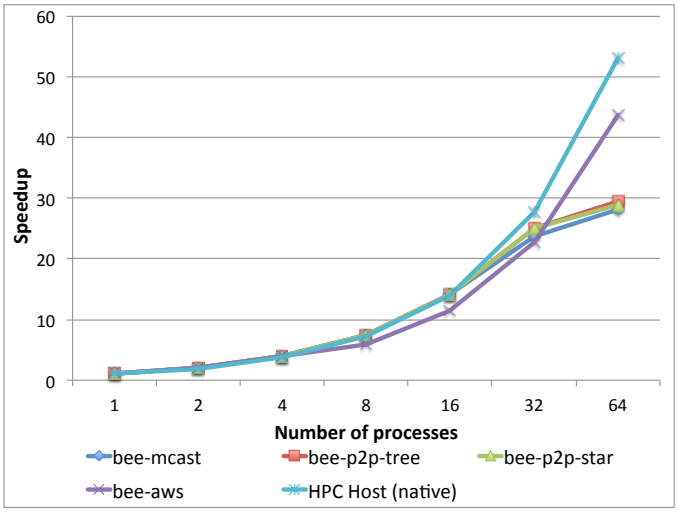


Figure 11: VPIC scale up test on BEE and HPC host



4 CASE STUDY

4.1 VPIC

In the section, we provide case study on BEE when running real HPC applications. We choose to test Vector Particle-In-Cell(VPIC) simulation tool [6–8] on BEE. VPIC is a general purpose particle-in-cell simulation tool for modeling kinetic plasmas in multiple spatial dimensions. VPIC is large scale parallel application that runs on multiple nodes using MPI and pthreads. It has optimzied for modern computing architectures by using short-vector, single-instruction-multiple-data (SIMD) instructions and cache optimization. Before the simulation begin, VPIC first need to load input deck and user configuration from files and write to output when done. Flexible checkpoint-restart semantics enabling VPIC checkpoint files to be read as input for subsequent simulations. 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 HPC systems and cloud computing system with its performance on bare metal HPC system. For the HPC system, we choose to use our testbed cluster system - Darwin. It has 16 Galton nodes, which have KVM enabled. Each node has two 8-core Intel Ivy Bridge E5-2650 v2 CPUs with 251GB RAM. For the cloud computing system, we choose to use Amazon Web Service. In order to get similar performance, we choose to use c3.4xlarge 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.

We test VPIC on different environment from using 1 process to 256 processes, which ranges from 1 computing node to 16 computing nodes. As shown in **Figure 11**, BEE-AWS solution scales similar to native host.

5 RELATED WORK

5.1 Shifter

Shifter [13] is an execution environment also aims to provide containerized environment for HPC systems. Since deploying standard Docker daemon on HPC systems imposes security and compatibility issue, they build a Docker-like container environment, which provides portability, isolation and reproducibility like Docker. The Shifter runs customized Shifter image. Docker users need to first import their Docker images and convert them into Shifter images before running. Shifter container can access 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 deployed on customized Cray machines with root privileges. On the other hand, BEE can run standard Docker images unmodified, which provides a higher usability for users. For example, developers can easily deploy consistent environment across their local development and test machines with production environment. Using standard Docker brings more convenience to share Docker image in Docker community. BEE has explicit application data management than can enable easy transfer across host machines for live migration or work flow integration. Also, BEE has several modes for data file sharing between processes that can be configured by user depends on whether file sharing is enabled on the hosts. If not, BEE can build its own file sharing mechanism, which brings more flexibility. Finally, BEE can be deployed on any HPC system and even cloud system without root privileges.

5.2 Singularity

Singularity [14] 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 image before running. It is also required to have root

privileges in order to deploy Singularity on a HPC system. But, unlike Shifter, it can be deployed on any HPC systems. It doesn't have a management on application data. Data sharing between containers also depends on the host machines. With multiple configuration solutions, BEE has more flexibility on deploying on HPC system. Besides providing an containerized execution environment, BEE bring better usability by combining data management, workflow integration, live migration, and cloud computing together to provide more convenient tool for HPC users and developers.

5.3 Charliecloud

Charliecloud [18] is a container solution that brings Docker-like environment into HPC system. It brings all the benefits of standard Docker container. The main benefit of this it that user or developers can have consistent building and execution environment across their local development or test machines to large scale production cluster machines. However, installing Charliecloud environment requires that the target HPC system has Linux kernel version to be least 3.18. It is challenging to install Charlieclound on current HPC system. It would take years before mainstream HPC system can upgrade Linux kernel that can meet the requirement of Charliecould. Also, it only manages the execution environment, however many other aspects of overall user workflow have not been considered.

5.4 AWS Container

Amazon EC2 Container Service(ECS) [1] is a Docker container service provided by Amazon on AWS. Since ECS deploys on top of EC2, and EC2 has a layer on virtual machines, ECS actually deploy Docker container layer on top of virtual machine layer, so basically it has the similar host-vm-docker structure as BEE. Regardless of the underlying hardware configuration, ECS provide a consistent building and execution environment by using standard BEE. Docker users can easily run their application on ECS without modification. That's why we deploy the similar structure on both cloud and HPC environment in BEE.

6 FUTURE WORK

In the next stage of this work, we will focus on several part.

(1) We will continue optimize the performance of BEE in terms of storage I/O, computing and network communication.
(2) We will continue to prefect our BEE framework. We will add more flexible and adaptive scheduling strategies and more sophisticated resource management system into BEE framework.
(3) Our current work can only utilize general CPUs, we will extend our BEE framework to utilizes accelerators such as GPU and Intel Xeon Phi, since accelerators are commonly used in HPC application.
(4) We will Integrate system level checkpoint/restoration capability into BEE. It can greatly benefit applications that do not native application level checkpoint/restoration capability built-in.

7 CONCLUSIONS

In this work, we first address the problems that current HPC users are facing. Then, we analysis the potential of integrating commonly used Docker container technology into the existing HPC system to build a consistent container environment for HPC users. Following this goal, we propose several goals that a successful containerized HPC framework should achieve and we design a Docker-enabled HPC framework – BEE. BEE achieves all the goals we set and also allows users to use the compute resources of both existing HPC system and cloud computing system with no extra configuration work. Experiments show that BEE can achieve comparable storage I/O performance and computing capability. Finally, case study on widely used VPIC simulation tool is tested on BEE and native HPC system for performance comparison.

REFERENCES

[1] Aws container service. Available on-line at: https://aws.amazon.com/ecs/.
[2] Docker. Available on-line at: https://www.docker.com/.
[3] Ior benchmark. Available on-line at: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/ior/.
[4] Packer. Available on-line at: https://www.packer.io/.
[5] BOETTIGER, C. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review 49*, 1 (2015), 71–79.
[6] BOWERS, K. J., ALBRIGHT, B., YIN, L., BERGEN, B., AND KWAN, T. Ultra-high performance three-dimensional electromagnetic relativistic kinetic plasma simulation a. *Physics of Plasmas 15*, 5 (2008), 055703.
[7] BOWERS, K. J., ALBRIGHT, B. J., BERGEN, B., YIN, L., BARKER, K. J., AND KERBYSON, D. J. 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), IEEE Press, p. 63.
[8] BOWERS, K. J., ALBRIGHT, B. J., YIN, L., DAUGHTON, W., ROYTERSHTEYN, V., BERGEN, B., AND KWAN, T. Advances in petascale kinetic plasma simulation with vpic and roadrunner. In *Journal of Physics: Conference Series* (2009), vol. 180, IOP Publishing, p. 012055.
[9] FOWLER, M., AND FOEMMEL, M. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* (2006), 122.
[10] GUGNANI, S., LU, X., AND PANDA, D. K. 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), ACM, pp. 36–45.
[11] HARJI, A. S., BUHR, P. A., AND BRECHT, T. Our troubles with linux kernel upgrades and why you should care. *ACM SIGOPS Operating Systems Review 47*, 2 (2013), 66–72.
[12] HUANG, W., LIU, J., ABALI, B., AND PANDA, D. K. A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international conference on Supercomputing* (2006), ACM, pp. 125–134.
[13] JACOBSEN, D. M., AND CANON, R. S. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group* (2015).
[14] KURTZER, G. M. Singularity 2.1.2 - Linux application and environment containers for science, Aug. 2016.
[15] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference, General Track* (2006), pp. 29–42.
[16] MATEESCU, G., GENTZSCH, W., AND RIBBENS, C. J. Hybrid computing?where hpc meets grid and cloud computing. *Future Generation Computer Systems 27*, 5 (2011), 440–453.
[17] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014), 2.
[18] PRIEDHORSKY, R., AND RANDLES, T. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. Tech. rep., Los Alamos National Laboratory (LANL), 2016.
[19] REUTHER, A., MICHALEAS, P., PROUT, A., AND KEPNER, J. Hpc-vms: Virtual machines in high performance computing systems. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on* (2012), IEEE, pp. 1–6.
[20] RUSSELL, R. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 95–103.
[21] TIKOTEKAR, A., VALLÉE, G., NAUGHTON, T., ONG, H., ENGELMANN, C., AND SCOTT, S. L. An analysis of hpc benchmarks in virtual machine environments. In *European Conference on Parallel Processing* (2008), Springer, pp. 63–71.
[22] VALLEE, G., NAUGHTON, T., ENGELMANN, C., ONG, H., AND SCOTT, S. L. System-level virtualization for high performance computing. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on* (2008), IEEE, pp. 636–643.

[23] ZHANG, J., LU, X., CHAKRABORTY, S., AND PANDA, D. K. D. Slurm-v: Extending slurm for building efficient hpc cloud with sr-iov and ivshmem. In *European Conference on Parallel Processing* (2016), Springer, pp. 349–362.