Using Container Migration for HPC Workloads Resilience

Mohamad Sindi
Center for Computational Engineering
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
sindimo@mit.edu

John R. Williams
Department of Civil and Environmental Engineering
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
jrw@mit.edu

***Abstract*—We share experiences in implementing a container-based HPC environment that could help sustain running HPC workloads on clusters. By running workloads inside containers, we are able to migrate them from cluster nodes anticipating hardware problems, to healthy nodes while the workloads are running. Migration is done using the CRIU tool with no application modification. No major interruption or overhead is introduced to the workload. Various real HPC applications are tested. Tests are done with different hardware node specs, network interconnects, and MPI implementations. We also benchmark the applications on containers and compare performance to native. Results demonstrate successful migration of HPC workloads inside containers with minimal interruption, while maintaining the integrity of the results produced. We provide several YouTube videos demonstrating the migration tests. Benchmarks also show that application performance on containers is close to native. We discuss some of the challenges faced during implementation and solutions adopted. To the best of our knowledge, we believe this work is the first to demonstrate successful migration of real MPI-based HPC workloads using CRIU and containers.**

Keywords—HPC fault tolerance, container migration, container performance benchmarks, MPI

I. Introduction

According to the November 2018 world's top 500 supercomputers list, 88.4% of the world's top high performance computing (HPC) systems are based on commodity hardware clusters [1]. Such systems are typically designed for performance rather than reliability. The Mean Time Between Failures (MTBF) for some of these top Petaflop systems has been reported to be several days [2], while studies estimate the MTBF for future Exascale systems to be less than 60 minutes [3]. Hence, running sustainable workloads on such systems is becoming more challenging as HPC systems grow.

In this paper, we share experiences in implementing and demonstrating a Linux container-based environment, which could improve the resilience of HPC workloads running on clusters. The environment is meant to help sustain running HPC workloads on the clusters in the event of anticipated node hardware issues that are common in HPC environments (e.g. ECC memory errors, etc.). The prediction or identification of such issues and silent errors is an area that has been previously investigated in the HPC domain and is not the scope of this paper, however it is addressed in another paper we currently have under publication review. The work presented in this paper on the other hand is a proposed remedy method to sustain running workloads once such issues are identified on the HPC system.

The method used to design our environment is based on running the HPC workloads inside Linux containers. The container technology has proven its success in the micro services domain (e.g. webservers) as a scalable and lightweight technology. In our work, we adapt the container technology towards HPC workloads to make use of its migration capabilities, which can be thought of as a fault tolerance mechanism. By running inside containers, we are capable of migrating the HPC workloads from nodes anticipating hardware problems, to healthy spare nodes while the workload is running. Migration is performed using the Checkpoint-Restore in Userspace (CRIU) tool [4] with no application modification. The container environment does not introduce any major interruption or performance overhead to the running workload. We also provide application benchmark results comparing the performance of the container environment to the native system.

The container environment is tested with various real HPC applications that are based on the Message Passing Interface (MPI) standard [5]. The applications tested come from both academia (open source) and the industry (closed source), and are written in various programming languages (e.g. C++, C, and Fortran). Verifying the integrity of the results produced by the migrated workloads is also addressed in this study. We test the applications with different hardware node types as well as different network interconnect types (e.g. 1, 10, and 25 Gigabit Ethernet networks). The applications are tested with various number of processing cores ranging from 4 and up to 144 cores. In addition, we test the migration with three commonly used MPI implementations, MPICH [6], Open MPI [7], and Intel MPI [8].

Our results demonstrate that we can successfully migrate real HPC workloads running inside containers from one physical machine to another with minimal interruption to the workload. The results produced from runs with and without a migration are identical with no data corruption. Our benchmark results also show that running these HPC applications inside containers provides a performance almost identical to the native system.

In this paper, we make the following contributions:

- The use of containers in the HPC domain as a fault tolerance enabling mechanism is a field that has not been thoroughly explored yet. To the best of our knowledge, we believe this work is the first in this domain to do successful migration of MPI-based real HPC workloads using CRIU and containers.
- We share several narrated YouTube videos to demonstrate the container migration of the various HPC workloads tested. This includes standard computational workloads, as well as workloads that produce in-situ visualization during the migration. We believe such video demonstration is also a first.
- Related work benchmarking containers typically use generic HPC benchmarks (e.g. HPL [9] and NPB [10]). This work benchmarks various real HPC applications. The benchmarks compare the performance of containers to native system and quantifies the overhead involved.
- As the use of containers in HPC is a young topic, we discuss some of the challenges faced during the study and share the solutions adopted to overcome them.

II. Related Work

Earlier studies investigated using virtual machine (VM) technologies to migrate workloads as a proactive fault tolerance measure [11], [12]. However, traditional VMs can introduce an overhead due to the virtual emulation of devices through a hypervisor. Earlier work showed that network latency with VMs can be more than twice in some cases compared to native [13], [14]. Nevertheless, VMs have been evolving with efforts to improve their performance using I/O bypass methods, such as the single root input/output virtualization (SR-IOV) specification [15]–[19]. However, such methods are limited to having a hardware/software setup that supports such features [20]. The performance with such methods is also still not competitive yet compared to native performance. Bypass methods may also affect the capability of VM migration. For example, the current Red Hat Enterprise Linux (RHEL) 7 documentation states that VM migration is not supported when SR-IOV is enabled [21].

Other studies looked into doing process-level migration where the actual MPI processes are targeted for migration. Study [22] attempted to use CRIU to perform such migration, however they were not successful in migrating parallel MPI processes and were able to only migrate a serial version of their application. Another tool targeted to checkpoint and migrate MPI processes is the Distributed MultiThreaded CheckPointing (DMTCP) package [23]. However, it requires the application binaries to be started through a proxy launcher, which may add an overhead for the application. In addition, it does not work with graphical applications using X-Windows extensions like OpenGL.

The current emerging alternative to VMs is the container technology. Compared to the well-explored domain of VM migration, container migration is a young topic that has not been extensively explored yet, especially in the scope of HPC. A few studies however have attempted container migration in Linux environments. Study [24] implemented a prototype libvirt-based custom Linux driver to enable container migration using CRIU. However, the study did not test migration with real applications, but rather with an artificially induced memory load on the system. Another study used a nonstandard tool called ReVirt [25] to migrate Docker containers using a logging/replay method [26]. The migrated application however was a standalone web application and not HPC related. A recent study looked into migrating containers using CRIU [27]. The migrated container did not have a real application in it either, but a simple serial Linux tool memhog [28], which just allocates memory on a system. Several studies have also compared the performance of containers to those of VMs and native systems [29]–[35]. However, most of these studies used generic benchmarks such as HPL or NPB, which are still valuable tests, but not truly reflective of real HPC applications.

III. System Design of Container Environment

A. *Cluster Setup*

Our cluster test environment was setup using the Amazon Web Services (AWS) infrastructure [36]. We believe that our container environment setup should also work with traditional Linux clusters that are not AWS based. By using AWS however, it gave us the option to setup test clusters with various specs. For our testing, we chose four types of node instances that varied in terms of specs [37]. Lower spec nodes were of type m4.2xlarge (4 physical cores each, 32 GB RAM, 1 Gig network). Medium spec nodes were of type c5.9xlarge (18 physical cores each, 72 GB RAM, 10 Gig network). As for the higher spec nodes, we tested using two types, m4.16xlarge (32 physical cores each, 256 GB RAM, 25 Gig network) and i3.metal (36 physical cores, 512 GB RAM, 25 Gig network). The i3.metal instances are AWS's new bare metal instances.

All instances access a shared network storage using Amazon's Elastic File System (EFS) [38]. The shared storage volume is mounted on the instances through the Network File System versions 4.1 (NFSv4.1) protocol [39]. This shared storage holds all the input and output data for the HPC applications.

B. *Software Setup*

In this section, we will discuss the software setup of our environment in terms of the operating system and container technology used, as well as the system tools used for launching and maintaining the containers on the system.

The container technology has been gaining a lot of attention in the past few years due to its low overhead and near native performance. Containers are more lightweight on the system compared to traditional VM technologies, as they do not rely on a hypervisor to access the underlying system resources. The container technology also provides a convenient way to package an application and its dependencies for ease of deployment.

There are several implementations of containers. For this work, we chose to use the OpenVZ containers [40]. OpenVZ provided out of the box container features such as assigning IP addresses to containers, secure shell (ssh) connections, and NFS shared file system mounting, which are all crucial features to any HPC environment. In addition, we use the CRIU tool to facilitate the container migrations. CRIU is an open source project that provides a checkpoint and restart functionality for Linux based applications and processes. The tool can be used to capture the states of CPU, memory, disk, and network of a process or a hierarchy of processes. Such functionality can also be applied to containers as they are considered Linux processes. We also use Parallels' "prlctl" command line tool to provision and manage the containers [41]. In addition, we installed the AWS Command Line Interface (CLI) package [42], which eased the management of the instances' networking part.

The OpenVZ version that we targeted for testing was 7.0.7. The kernel provided by OpenVZ is a variation based on Red Hat Enterprise Linux (RHEL) 7.4 kernel 3.10.0-693. The OpenVZ variation however has patches that support the container functionally and its management. To our understanding, the patches applied to the kernel are currently not mainstream in the Linux kernel. The CRIU version used was version 3.4.0.41-1, while the "prlctl" container management tool used was version 7.0.148-1. The AWS CLI package used was version 1.15.61.

It is also important to note that the only officially supported method to install OpenVZ is through their ISO installation image. This created a challenge to us as our testing environment is setup on the AWS platform and there was no official way to install an instance with an ISO image. Having talked with the OpenVZ community, it was brought to our attention that there is a commercially supported version of OpenVZ, called Virtuozzo, which AWS currently provides instances for. Due to the limitation of not being able to install via ISO image on AWS, we ended up setting our environment using the equivalent Virtuozzo 7.0.7 image provided by AWS. Technically, the kernel and software versions of CRIU and the "prlctl" container management tool for both distributions appeared identical to us.

C. *Container Setup*

HPC applications typically would fully utilize the underlying compute node they are running on. With this in mind, our setup consisted of launching a single container on each node, with full access to the underlying resources. By default, the launched containers do not have access restrictions in terms of how many processing cores to use from the underlying machine. However when it comes to memory, we had to specify the amount of memory we wanted to allocate for the container. We set the containers to use the majority of the memory available, while leaving a sufficient amount of memory for the underlying operating system just as a precaution measure. For example, on the i3.metal instances, the containers were launched with 500 GB of RAM out of the 512 GB available.

We also wrote scripts to automate creating and launching the containers with specific settings using the "prlctl" tool. This included setting the container's operating system template, name, IP address, users, DNS, NFS mounts, ssh keys, default bash shell environment variables, default bash shell limits, and the netfilter firewall settings. The container template used was the CentOS 7 template. In addition, we used the "vzpkg" tool [43] to automate installing any extra packages we needed inside the container (e.g. MPI libraries, third party packages needed by the HPC applications being tested, etc.). For each of the HPC applications we were testing, a separate container installation image was created. Each image only included the software packages and application binaries needed for that HPC application. This kept the containers minimal in size.

D. *Network Setup*

For every instance in the cluster, we allocated four IP addresses. The native instance was assigned 2 IPs, one is public and one is private. The container running on the instance was also assigned 2 IPs, one is public and one is private. The public IPs are used to access the native instance and the container by the ssh protocol from any remote terminal. The private IPs are used for local communication between the instances or containers. The IPs of the native system are associated with the instance's primary network interface, while the IPs of the container are associated with the instance's secondary network interface. By having this setting, the IPs of the containers can be treated as floating IP addresses, meaning that if a container is migrated from one physical machine to another, then its IP address can also be migrated along with it. The IPs were set to be static as it was more convenient when doing our benchmarking and container migration testing. The details of how the containers' IPs were setup can be found in this reference [44].

IV. HPC Applications to Test

For testing, we chose six HPC applications that are all MPI based and cover various domains. The applications tested were the Ohio State University (OSU) Micro-Benchmarks version 5.3.2 [45], Palabos version v2.0r0 [46], Flow version 2018.04-0 [47], Fluidity version 4.1.15 [48], GalaxSee version MPI 0.9 [49], and the ECLIPSE* industry-reference reservoir simulator by Schlumberger version 2017.2 [50]. Table 1 summarizes these applications.

TABLE 1. Summary of tested HPC applications

Application	Main Developers	Language	Domain
OSU Micro-Benchmarks	Ohio State University	C	MPI benchmark for network bandwidth and latency
Palabos	- Academic: University of Geneva - Industry: FlowKit CFD	C++	CFD/Complex Physics using lattice Boltzmann method
Flow	Open Porous Media Initiative (Oil companies + Academia)	C++	Reservoir simulation
Fluidity	Imperial College London	Fortran, C++	- CFD solving Navier-Stokes - Geophysical fluid dynamics - Ocean Modelling - Adaptive unstructured mesh
GalaxSee	- Shodor Education Foundation - National Center for Supercomputing Applications - George Mason University	C++	N-body galaxies simulation
ECLIPSE	Schlumberger (commercial)	Fortran	Industry-reference reservoir simulator

V. Testing and Results

A. *Performance Benchmarks on Containers*

The first type of tests conducted was to benchmark the HPC applications on the container environment and compare the performance to the native system. A native system is a cluster of instances that have no containers running on them. A container-based environment would be the same cluster of instances, however on each instance we launch a single container that encapsulates the entire underlying native system. The native instances and the containers have their own distinct IP addresses. The mpirun launcher is then used to start the MPI jobs. The MPI hostfile will have the appropriate IP addresses of either the native instances or the containers. We record the elapsed time of execution for both native and container runs. Each benchmark is ran four times and timings are averaged. We also calculate the overhead involved with containers. All benchmarks were performed using the MPICH MPI library version 3.0-3.0.4-10, except for the ECLIPSE simulator where we used the Intel MPI library version 5.0.2.044. Intel MPI was the supported library for the closed-source binary executable of the ECLIPSE simulator.

The first application tested was the OSU Micro-Benchmarks. This application is the only one out of the six we tested that is not a real HPC application, but rather a network performance benchmark for typical MPI point-to-point and collective operations. It allowed us to compare that container's network performance to that of the native system in terms of latency and bandwidth. For the point-to-point benchmarks, we tested osu_latency and osu_bw. For the collective benchmarks, we tested osu_allgather and osu_allreduce, which are two commonly used collective MPI operations in HPC applications. Each benchmark was performed on a pair of instances of the same type. In the case of the point-to-point benchmarks, one processor from each instance participated in the benchmark, while for the collective benchmarks all available processing cores on each instance were used. As our instances vary in network specs, the tests were performed on all three network types available, which included 1, 10, and 25 Gigabit networks. Fig. 1 shows the point-to-point benchmarks. Fig. 2 shows the collective benchmarks.

In the case of the 1 Gigabit network, the bandwidth of the container and native system were almost identical. With the 10 Gigabit network, we noticed a slight bandwidth reduction on the containers, while on the 25 Gigabit network we saw a more noticeable reduction in bandwidth. Nevertheless, the overall average bandwidth overhead was around 3.9 % with containers. As for latency, there was a slight overhead with containers for most of the tests. Overall, the average latency overhead was around 6.8 % when using the container environment. We do not fully understand why the 25 Gigabit network behaved a bit differently than the others, but we believe this might be related to the need for more appropriate network tuning for 25 Gigabit networks from the Linux side (e.g. kernel parameters, etc.).

Such slight overhead with containers was to be expected. We believe that the difference in performance was due to the way the network routing is done. The container's network is run in a host-routed mode. In this case, the native node acts as a router to the packets passing through the container. Hence, the container's traffic will have to go through that one extra step of routing, which contributes to the slight overhead observed.

Next, we proceeded with testing the remaining HPC applications. The benchmarks were performed on the various instance types with MPI job sizes ranging from running on 4 processing cores, and up to 144 cores. The input models we used for each application varied depending on the number of processing cores we were running on. For example, when testing the Fluidity application, we used a moderately sized benchmark model "tephra_settling" when running on 4 cores, however when benchmarking on 144 cores, we used a significantly larger model "tides_in_the_Mediterranean_Sea". Also note that for the ECLIPSE simulator and Flow applications, we were only able to test on up to 8 processing cores. For the ECLIPSE simulator, the commercial license we had was limited to only 8 parallel processes. For Flow, the test models provided with the application were limited in size (e.g. Norne model was the largest) and problem sizes were too small to be run on more than 8 cores. Table 2 summarizes all input models used.

TABLE 2. INPUT MODELS USED FOR APPLICATION TESTING

Instance Type	Application	Model	# Nodes	# Cores
m4.2xlarge	Fluidity	tephra_settling	1	4
m4.2xlarge	Fluidity	water_collapse	2	8
c5.9xlarge	Fluidity	water_collapse	1	18
c5.9xlarge	Fluidity	backward_facing_step_3d	2	36
m4.16xlarge	Fluidity	tides_in_the_Mediterranean_Sea	4	128
i3.metal	Fluidity	tides_in_the_Mediterranean_Sea	4	144
m4.2xlarge	Palabos	rayleighTaylor2D_2000x600	1	4
m4.2xlarge	Palabos	rayleighTaylor2D_2000x600	2	8
c5.9xlarge	Palabos	rayleighTaylor2D_2000x600	1	18
c5.9xlarge	Palabos	rayleighTaylor2D_4000x1200	2	36
m4.16xlarge	Palabos	rayleighTaylor2D_8000x2400	4	128
i3.metal	Palabos	rayleighTaylor2D_8000x2400	4	144
m4.2xlarge	GalaxSee	#stars: 8000, mass: 10, time: 10000	1	4
m4.2xlarge	GalaxSee	#stars: 8000, mass: 10, time: 10000	2	8
c5.9xlarge	GalaxSee	#stars: 18000, mass: 10, time: 10000	1	18
c5.9xlarge	GalaxSee	#stars: 36000, mass: 10, time: 10000	2	36
m4.16xlarge	GalaxSee	#stars: 38400, mass: 10, time: 1000	4	128
i3.metal	GalaxSee	#stars: 43200, mass: 10, time: 1000	4	144
m4.2xlarge	Flow	SPE9	1	4
m4.2xlarge	Flow	Norne	2	8
m4.2xlarge	ECLIPSE	ONEM1_4	1	4
m4.2xlarge	ECLIPSE	ONEM1_8	2	8

Fig. 3 shows the average elapsed times for the benchmarks comparing the performance of container to native. Even though we previously saw a slight overhead in latency and bandwidth on containers with the OSU Micro-Benchmarks, the overhead when testing with real HPC application was very negligible and performance was close to native (0.034% average performance overhead). In the worst case, the maximum overhead observed was around 0.9%. Overall, the performance on the container environment was acceptable for all applications.

B. Testing Container Migration

a) Migration Mechanism: Our goal was to launch a distributed MPI application inside containers that are hosted on several native machines, then try to successfully migrate one of the containers from one native machine to another native machine (i.e. spare machine) while the MPI job was running. Fig. 4 gives an overview of the task we were trying to accomplish. We used the "prlctl" tool as our interface to control the states of the container. The "prlctl" tool uses the CRIU library to manipulate the container's state. The AWS CLI tool is also used to migrate the container's floating IP address between machines.

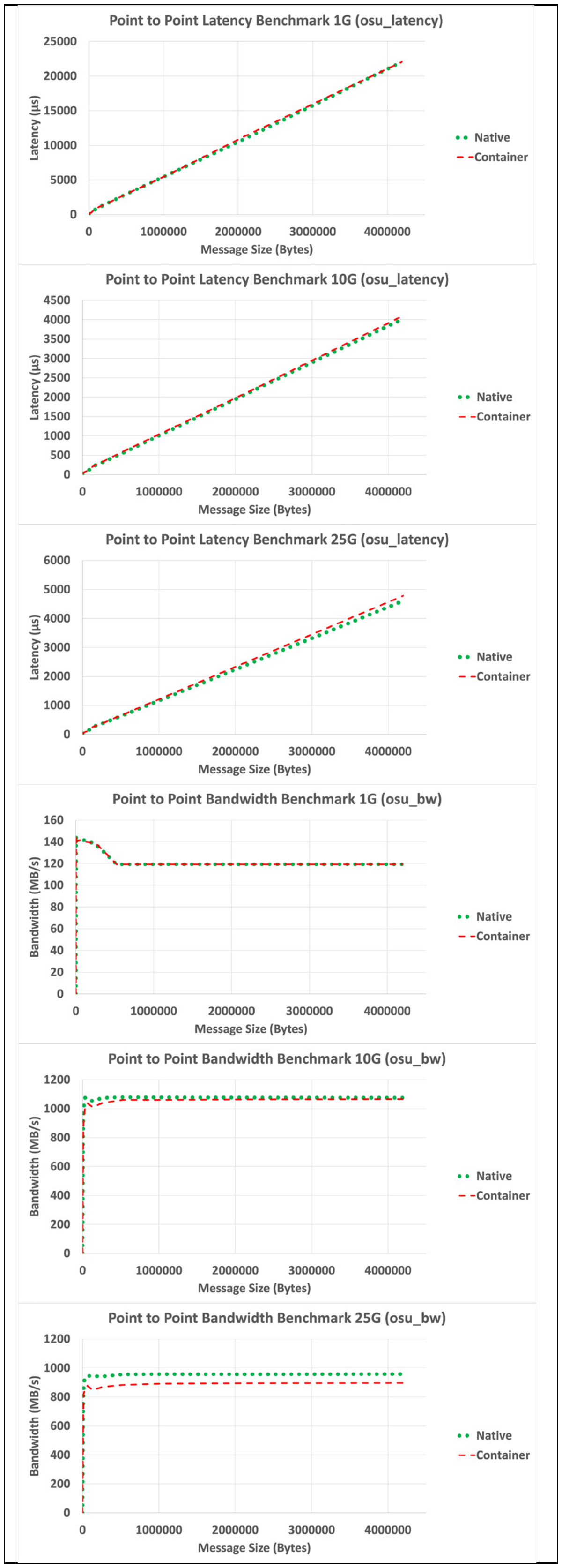

Fig. 1. Point-to-Point MPI benchmarks

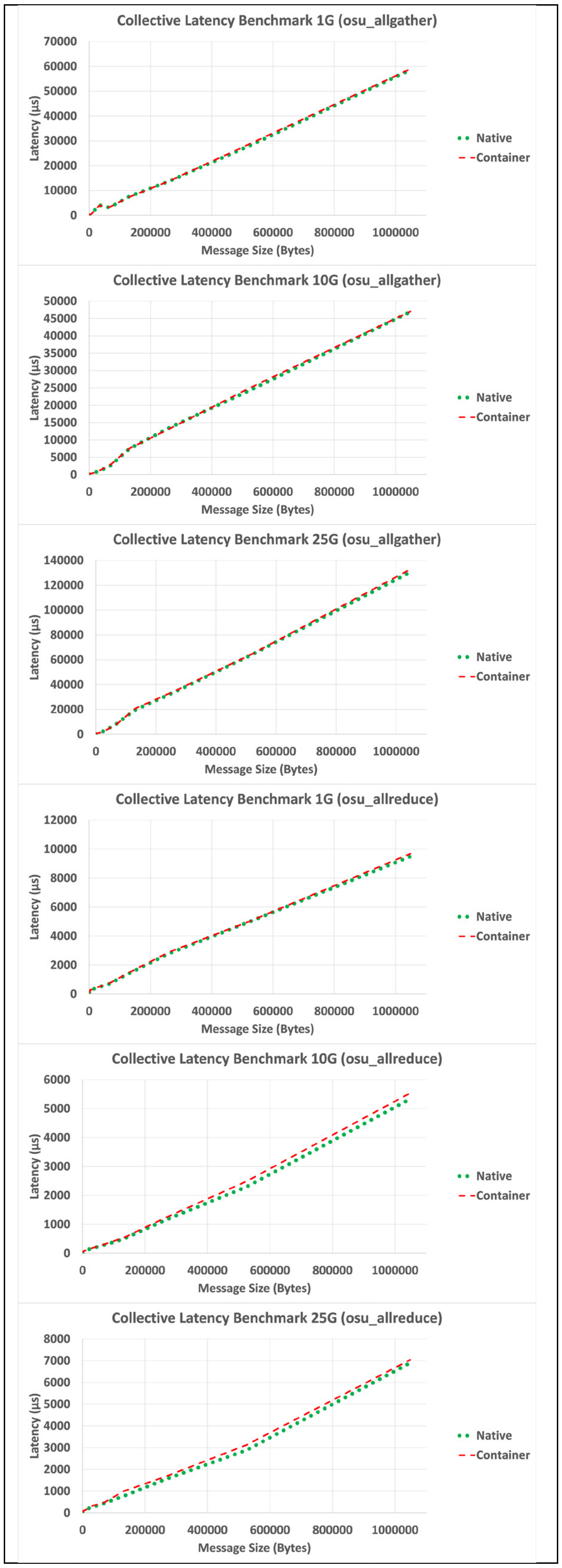

Fig. 2. Collective MPI benchmarks

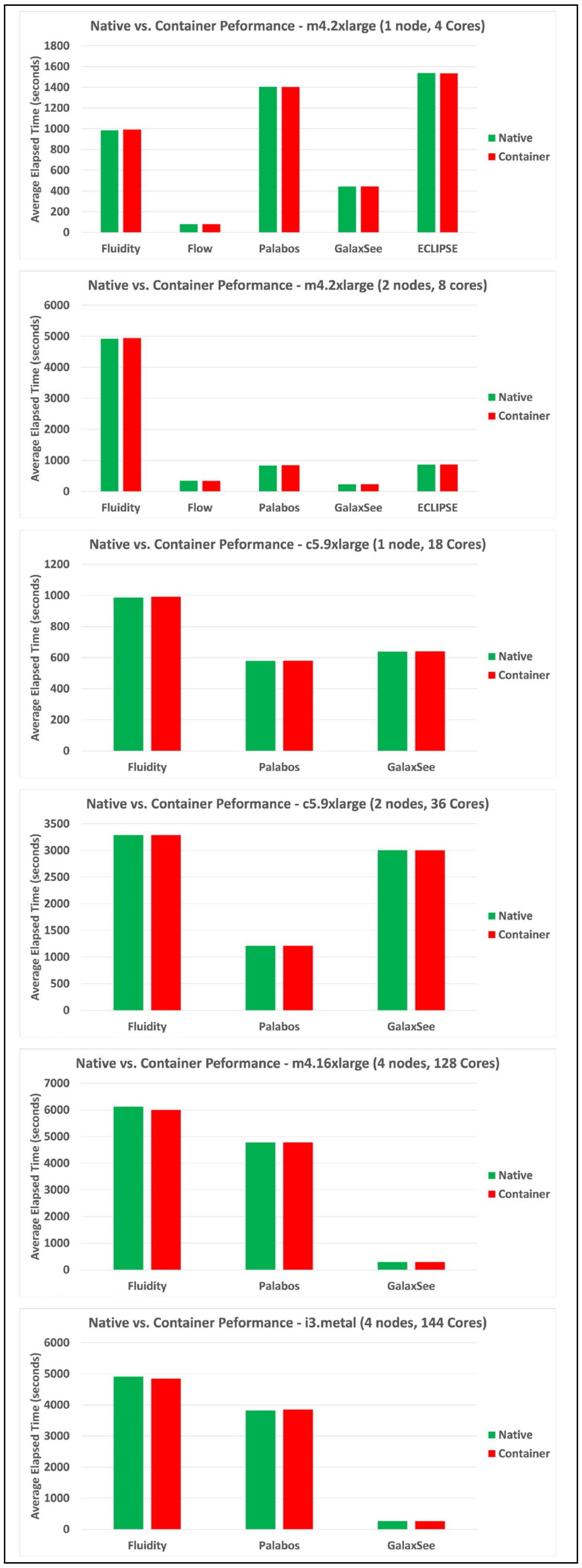

Fig. 3. Application benchmarks on various instance types

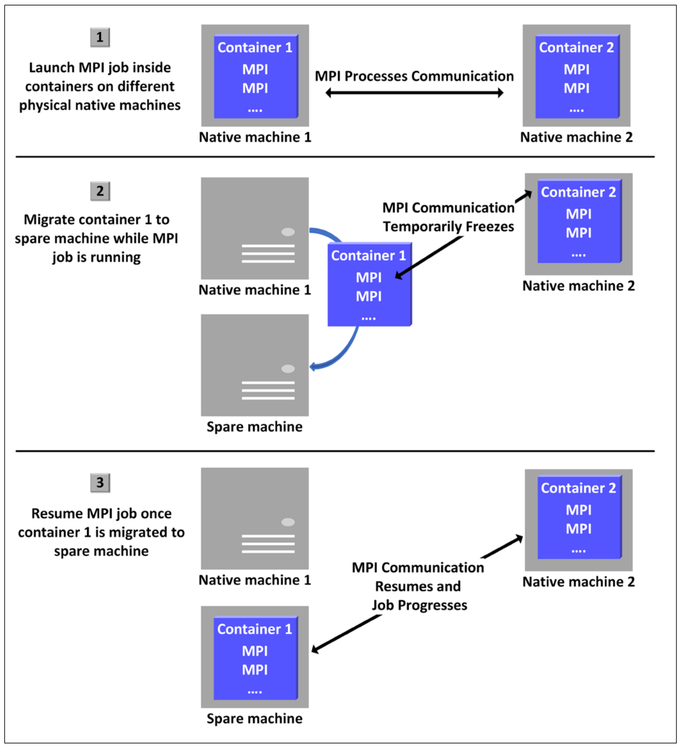
1
Launch MPI job inside containers on different physical native machines
Container 1
MPI
MPI
Native machine 1
MPI Processes Communication
Container 2
Native machine 2
2
Migrate container 1 to spare machine while MPI job is running
MPI Communication Temporarily Freezes
Spare machine
3
Resume MPI job once container 1 is migrated to spare machine
MPI Communication Resumes and Job Progresses

Fig. 4. Overview of container migration test

Overall, the migration steps can be described as follows. First, the container is put in a "suspend" state as MPI processes are running inside of it. This will cause the MPI job to freeze temporarily. The MPI processes on the other participating nodes will still be alive, but waiting for the suspended container to come back. As part of the suspension process, CRIU takes a dump of the container's state and stores it locally. Next, the container's floating IP address gets migrated to the spare node. We automated the IP migration by writing a custom script influenced by Sabat's [51], which utilizes the AWS CLI tool for automation. After that, the container is put in a "migrate" state, which will copy the dumped container's state to the spare machine. Finally, the container is put in a "resume" state on the spare machine. At this point, the MPI job resumes from the same state before it froze and progresses. The migration and resuming is done as user root. We automate the entire migration process with a custom script. Fig. 5 summarizes the migration steps.

b) Migration Time: We also observed the time it takes to complete the container migration for the various applications tested. The migration time was influenced by the size of application binaries and dependent libraries stored inside container. In the case of Palabos, GalaxSee, and the ECLIPSE simulator, those files were stored on a shared NFS storage and not inside container, hence making the container's size relatively smaller. In the case of Fluidity and Flow, the installation of the applications was performed using RPM packages. This method ends up installing the application binaries and their dependencies inside the container, thus increasing the size of the container.

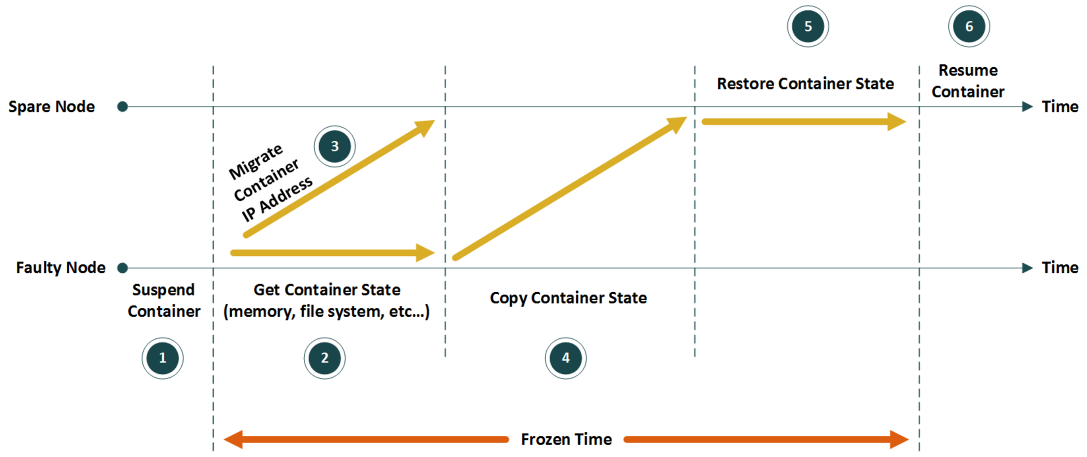

Fig. 5. Container migration steps

We also noticed that the migration time did not change much either we were using the 1, 10, or 25 Gigabit network. It seemed to us later that the bottleneck was not the network speed, but rather the read/write speeds of the local disks storing the container's dumped data. The instances by default use a General Purpose SSD (gpt2) local disk that has a limited throughput. AWS also provides higher throughput disks called Provisioned IOPS SSD (io1). We tested migration with both disks and the average migration time improved by around 35% when using the io1 disks. The average migration time with the gpt2 disks was 34 seconds, while with the io1 disks it was 22 seconds. Table 3 shows the average container migration times for the different applications using the two types of disks.

TABLE 3. AVERAGE CONTAINER MIGRATION TIMES

Application	Migration Time gpt2 Disks (seconds)	Migration Time io1 Disks (seconds)
Fluidity	50	33
Flow	35	23
Palabos	30	19
GalaxSee	29	19
ECLIPSE	26	17

C. *Integrity Check for Produced Results*

As part of the container migration testing, it was crucial to check the integrity of the results produced and make sure that the migration was not causing any data corruption. The different HPC applications tested produced several types of output files. Fluidity produces binary files (e.g. .vtu and .pvtu files) for post processing with ParaView [52]. Palabos also produces binary files for post processing with ParaView (e.g. .vtk files), as well as GIF image files. The ECLIPSE simulator and Flow produce text files (e.g. .log and .prt files), as well as binary output files for post processing (e.g. .unsmry and .egrid files). GalaxSee does not generate output files, however it produces an in-situ visualization.

We run each application with and without migration and do a comparison of the results using the relevant post processing tools, and also compare output text files. Fig. 6 is an example demonstrating the data integrity comparison check for a Palabos simulation of the Rayleigh Taylor instability. The container migration was triggered randomly at time step 2400 during the run. We compare results produced with and without container migration at time step 2400, and then at time step 3000 after the migration completed and the simulation has progressed, and lastly at the final time step. Both results were the same. None of applications tested had any discrepancy issues with the results.

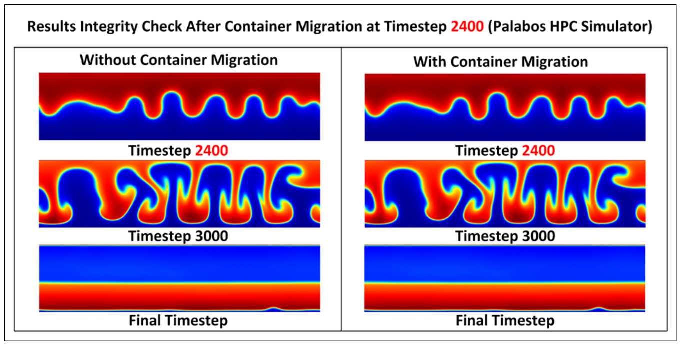

Fig. 6. Results integrity check for Palabos simulation

D. *Container Demo Videos*

In this section, we share several YouTube videos demonstrating the successful container migration for some of the HPC applications we tested. We describe the individual tests and reference the links for the videos. The videos are visually narrated to have a better understanding of the test scenarios.

The test environment is the same as illustrated in Fig. 4. During the test, the Linux "top" command is run inside each container to monitor the state of the MPI processes. On the native machines, we use the Linux "wall" command to run the "prlctl" tool every second to list the containers' states. This helps visually catch the transition states of the containers during migration. All demos were done using instances with gpt2 disks. Table 4 references the links for the demo videos.

The first video demonstrates the container migration for Palabos [53]. The application is launched on two containers with two MPI processes on each. The run is for a 2D simulation of the Rayleigh Taylor instability. It shows a 2D in-situ visualization of the results as it runs. Container "sindi_ct1" is then migrated at a random time step 2400. During migration, the state of the container changes from “running” to “suspended” and the MPI job freeze temporarily. Once the container is migrated to the spare machine, the state of the container changes back to “running” and the application’s time stepping and 2D visualization resumes.

The second video demonstrates the container migration for the ECLIPSE simulator [54]. The job is launched on two containers with four MPI processes each. The simulation is run using the “PARALLEL_15000CELLS” model. We were curious how the job behaved if the container was migrated more than once. Hence, we migrate it back and forth between two native machines. The job progresses successfully during both migrations.

The third video demonstrates the container migration for GalaxSee [55]. It is launched on two containers with four MPI processes on each. The simulation is done for eight thousand stars and an in-situ visualization is displayed. We swap the two containers, sindi_ct1 and sindi_ct2, simultaneously as the job runs. We did this test because we were curious how the native machines would behave during a simultaneous migration. During the swap, the visualization temporarily freezes then resumes. Having two simultaneous migrations increased the average migration time to 38.5 seconds instead of 29 seconds.

The last demo is for the Flow application [56]. This test does not involve container migration, but rather further tests the container’s suspend and resume capabilities. The job is launched on two containers with four MPI processes each. Next, we suspend all the containers involved in the job, which causes the job to freeze. After that, we power off the two native machines hosting the two containers and then power them on again. This is done by issuing the “reboot” command on both machines. Once the machines are up again, we resume both containers. The job then resumes and progresses. This demo shows that the container’s resilience potentials are not only limited to migrating containers. The ability to suspend an entire MPI job then resuming it can be useful for HPC centers, especially in events such as scheduled system downtime maintenance.

TABLE 4. VIDEO LINKS FOR CONTAINER DEMOS

Application	**Demo Video Link**
Palabos	https://youtu.be/1v73E2Ao3Mk
ECLIPSE	https://youtu.be/5tz6JP2UgTk
GalaxSee	https://youtu.be/NlT7nJ-yENc
Flow	https://youtu.be/KNTVHQnMVHU

VI. CHALLENGES AND SOLUTIONS

We faced several technical challenges throughout testing containers in an HPC environment. In this section, we go over some of the challenges encountered and the solutions adopted.

MPICH, and its derivatives, is considered the most widely used implementation of MPI in the world according to its website [57]. While testing with MPICH and its derivative Intel MPI, we did not have technical issues in terms of launching MPI jobs on containers, nor when it came to migrating the containers. On the other hand, with Open MPI the launching of an MPI job inside the containers was failing. We used Open MPI version 1.10.7, which was the current during testing. The jobs would fail to start with the following error “tcp_peer_send_blocking: send() to > socket 9 failed: Broken pipe”. At the time of this study, the Open MPI frequently asked questions (FAQ) page mentions that Open MPI does not to support virtual IP interfaces [58]. OpenVZ uses such interfaces inside the containers (i.e. venet0 and venet0:0). Nevertheless, we used a hack to the Open MPI source code to overcome the issue [59]. It is however a workaround with no official support.

Another issue encountered was related to having a shared NFS storage mounted inside the containers. When attempting to migrate a container with an active NFS mount, the migration process immediately hanged. This took place in the first step of the migration process when suspending the container. After investigation, we were able to pinpoint that the issue was related to the CRIU library. CRIU was hanging during the execution of the code in one of its files “nfs-ports-allow.sh”. Debugging that code further, we found it was hanging in the "nfs_server_ports" function. That function tries to run the Linux “rpcinfo” tool to remotely query open NFS ports on the shared storage system mounted inside the containers, which was causing the hanging. We implemented a fix to CRIU’s original code to overcome this issue. Instead of relying on the “rpcinfo” tool, we replaced that code with a variable that references the appropriate standard NFS port numbers. For example, in our case the storage mounts were using the NFSv4.1 protocol, so the standard corresponding port number was port 2049. After applying this fix to CRIU, we were able to migrate the containers.

The last problem we had was specific to the ECLIPSE simulator. Initially, the container migration was failing in the suspension stage. Investigating this further, logs where showing the message “remote posix locks on NFS are not supported yet”. Apparently, CRIU currently has a limitation of not being able to suspend processes having file locks [60]. The ECLIPSE simulator was producing a database file “.dbprtx” with a file lock. Release notes of the ECLIPSE simulator showed that this file is optional and that it might cause issues on shared file systems that do not support file locking [61]. Fortunately, there was an option to turn off the generation of this file and we were able to migrate the containers afterwards.

VII. CONCLUSION AND FUTURE WORK

In this paper we presented our experiences in using CRIU container migration to help improve the resilience of running HPC workloads on clusters. We tested the container’s migration capabilities using various real HPC applications. Results show that we can successfully migrate containers with HPC workloads between different physical machines, with minimal interruption and no data corruption. We also did a broad range of benchmarks on containers using real HPC applications. This included benchmarking using three interconnect types and four different machine types varying in specs. Results show that the performance on containers was close to native. For future work, we plan to investigate container migration in HPC environments having InfiniBand networks. We will also look into testing CRIU container migration with different container types such as Singularity and Docker.

REFERENCES

[1] The TOP500 Project, "TOP500 List Statistics," Nov-2018. [Online]. Available: http://www.top500.org/statistics/list.

[2] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[3] R. Riesen *et al.*, "Redundant computing for exascale systems," *Sandia National Laboratories*, 2010.

[4] Virtuozzo, "Checkpoint-Restore in Userspace (CRIU)." [Online]. Available: https://www.criu.org.

[5] D. W. Walker and J. J. Dongarra, "MPI: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[6] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, "Users' Guide to mpich, a Portable Implementation of MPI," *Argonne National Laboratory*, vol. 9700, pp. 60439–4801, 1995.

[7] E. Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2004, pp. 97–104.

[8] Intel, "Intel MPI Library." [Online]. Available: https://software.intel.com/en-us/mpi-library.

[9] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[10] D. H. Bailey, "NAS parallel benchmarks," *Encyclopedia of Parallel Computing*, pp. 1254–1259, 2011.

[11] A. Polze, P. Troger, and F. Salfner, "Timely virtual machine migration for pro-active fault tolerance," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, 2011, pp. 234–243.

[12] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *Proceedings of the 21st annual international conference on Supercomputing*, 2007, pp. 23–32.

[13] J. P. Walters and V. Chaudhary, "A fault-tolerant strategy for virtualized HPC clusters," *The Journal of Supercomputing*, vol. 50, no. 3, pp. 209–239, 2009.

[14] B. Davda and J. Simons, "RDMA on vSphere: Update and future directions," in *Open Fabrics Workshop*, 2012.

[15] P. Kutch, "Pci-sig sr-iov primer: An introduction to sr-iov technology," *Intel application note*, pp. 321211–002, 2011.

[16] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[17] J. Jose, M. Li, X. Lu, K. C. Kandalla, M. D. Arnold, and D. K. Panda, "SR-IOV support for virtualization on infiniband clusters: Early experience," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 385–392.

[18] M. Musleh, V. Pai, J. P. Walters, A. Younge, and S. Crago, "Bridging the virtualization performance gap for HPC using SR-IOV for InfiniBand," in *2014 IEEE 7th International Conference on Cloud Computing*, 2014, pp. 627–635.

[19] 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, pp. 125–134.

[20] Mellanox, "HowTo Configure SR-IOV for ConnectX-3 with KVM (InfiniBand)." 05-Dec-2018.

[21] RedHat, "SR-IOV Support For Virtual Networking." 05-Jan-2018.

[22] A. Reber, "Process migration in a parallel environment." Universität Stuttgart, 23-Jun-2016.

[23] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–12.

[24] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, "Migrating LinuX containers using CRIU," in *International Conference on High Performance Computing*, 2016, pp. 674–684.

[25] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[26] C. Yu and F. Huan, "Live migration of docker containers through logging and replay," in *2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*, 2015.

[27] R. Stoyanov and M. J. Kollingbaum, "Efficient Live Migration of Linux Containers," in *High Performance Computing*, vol. 11203, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 184–193.

[28] A. Kleen, "memhog: Allocation of memory with policy for testing," 2019. [Online]. Available: http://man7.org/linux/man-pages/man8/memhog.8.html.

[29] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, pp. 275–287.

[30] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, 2015, pp. 171–172.

[31] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013, pp. 233–240.

[32] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance evaluation of containers for HPC," in *European Conference on Parallel Processing*, 2015, pp. 813–824.

[33] J. Zhang, X. Lu, and D. K. Panda, "High performance MPI library for container-based HPC cloud on InfiniBand clusters," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 268–277.

[34] 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 *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1777–1784.

[35] J. Zhang, X. Lu, and D. K. Panda, "Is Singularity-based Container Technology Ready for Running MPI Applications on HPC Clouds?," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017, pp. 151–160.

[36] Amazon, "Amazon Web Services (AWS)," 2019. [Online]. Available: https://aws.amazon.com.

[37] Amazon Web Services, "Amazon EC2 Instance Types," 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types.

[38] Amazon Web Services, "Amazon Elastic File System (EFS)," 2019. [Online]. Available: https://aws.amazon.com/efs.

[39] S. Shepler, M. Eisler, and D. Noveck, "Network file system (NFS) version 4 minor version 1 protocol," 2010.

[40] M. Furman, *OpenVZ essentials*. Packt Publishing Ltd, 2014.

[41] OpenVZ, "OpenVZ Command Line Reference: prlctl." [Online]. Available: https://docs.openvz.org/openvz_command_line_reference.webhelp/_prlctl.html. [Accessed: 01-Apr-2019].

[42] Amazon, "AWS Command Line Interface (CLI)." [Online]. Available: https://aws.amazon.com/cli. [Accessed: 01-Apr-2019].

[43] OpenVZ, "OpenVZ Command Line Reference: vzpkg." [Online]. Available: https://docs.openvz.org/openvz_command_line_reference.webhelp/_vzpkg.html. [Accessed: 01-Apr-2019].

[44] Virtuozzo, "Using Virtuozzo in the Amazon EC2." [Online]. Available: https://docs.virtuozzo.com/wiki/Using_Virtuozzo_in_the_Amazon_EC2

#Configure_the_external_IP_address_for_the_container. [Accessed: 10-Jan-2018].

[45] Ohio State University, "OSU Micro-Benchmarks," 2018. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks. [Accessed: 01-Apr-2018].

[46] J. Latt, "Palabos, Parallel Lattice Boltzmann Solver," *FlowKit, Lausanne, Switzerland*, 2009.

[47] Open Porous Media (OPM) Initiative, "Flow: Fully-Implicit Black-Oil Simulator," 2018. [Online]. Available: https://opm-project.org/?page_id=19. [Accessed: 27-Mar-2018].

[48] Imperial College London AMCG, "Fluidity manual v4.1.12." 22-Apr-2015.

[49] D. Joiner, "GalaxSee-MPI: Gravitation N-Body Simulation." [Online]. Available: https://www.shodor.org/refdesk/Resources/Tutorials/MPIExamples/GalaxSee.php. [Accessed: 04-Mar-2018].

[50] Schlumberger, "ECLIPSE Industry-Reference Reservoir Simulator, * Mark of Schlumberger," 2017. [Online]. Available: https://www.software.slb.com/products/eclipse.

[51] T. Sabat, "AWS CLI Script to Assign a Secondary IP," 09-Jun-2015. [Online]. Available: https://codepen.io/tsabat/post/aws-cli-script-to-assign-a-secondary-ip.

[52] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large data visualization," *The visualization handbook*, vol. 717, 2005.

[53] M. Sindi, "Container Migration Demo for MPI-based HPC Workloads: Palabos Application," Massachusetts Institute of Technology, 2019. [Online]. Available: https://youtu.be/1v73E2Ao3Mk.

[54] M. Sindi, "Container Migration Demo for MPI-based HPC Workloads: The ECLIPSE Simulator by Schlumberger," Massachusetts Institute of Technology, 2019. [Online]. Available: https://youtu.be/5tz6JP2UgTk.

[55] M. Sindi, "Container Migration Demo for MPI-based HPC Workloads: GalaxSee Application," Massachusetts Institute of Technology, 2019. [Online]. Available: https://youtu.be/NlT7nJ-yENc.

[56] M. Sindi, "Container Demo for MPI-based HPC Workloads: SuspendResume Entire MPI Job - Flow Application," Massachusetts Institute of Technology, 2019. [Online]. Available: https://youtu.be/KNTVHQnMVHU.

[57] W. Gropp, L. Ewing, N. Doss, and A. Skjellum, "MPICH High-Performance Portable MPI." [Online]. Available: https://www.mpich.org. [Accessed: 21-Mar-2017].

[58] Open MPI, "Open MPI FAQ: Does Open MPI support virtual IP interfaces?," 22-Jan-2019. [Online]. Available: https://www.open-mpi.org/faq/?category=tcp#ip-virtual-ip-interfaces. [Accessed: 03-Apr-2019].

[59] J. Squyres, "Open MPI Hack to Work with OpenVZ Containers," 24-Jun-2016. [Online]. Available: https://www.mail-archive.com/users@lists.open-mpi.org/msg29585.html. [Accessed: 04-Dec-2017].

[60] CRIU, "What cannot be checkpointed." 13-Jul-2017.

[61] Schlumberger, "ECLIPSE Industry-Reference Reservoir Simulator (Version 2017.2) - Release Notes." 2017.