HPC Containers in Use

Jonathan Sparks
SC and Clusters R&D
Cray Inc.
Bloomington, MN, USA
e-mail: jsparks@cray.com

Abstract— **Linux containers in the commercial world are changing the landscape for application development and deployments. Container technologies are also making inroads into HPC environments, as exemplified by NERSC's Shifter and LBL's Singularity. While the first generation of HPC containers offers some of the same benefits as the existing open container frameworks, like CoreOS or Docker, they do not address the cloud/commercial feature sets such as virtualized networks, full isolation, and orchestration. This paper will explore the use of containers in the HPC environment and summarize our study to determine how best to use these technologies in the HPC environment at scale.**

Keywords- Shifter; HPC; Cray; container; virtualization; Docker; CoreOS

I. Introduction

Containers seem to have burst onto the infrastructure scene for good reason. Due to fast startup times, isolation, a low resource footprint, and encapsulation of dependencies, they are rapidly becoming the tool of choice for large, sophisticated application deployments. To meet the increasing demand for computational power, containers are now being deployed on HPC systems.

Virtualization technologies have become very popular in recent years, such as Xen [1] and KVM [2]. These hypervisor-based virtualization solutions bring several benefits, including hardware independence, high availability, isolation, and security. They have been widely adopted in industry computing environments. For example, Amazon Elastic Cloud (EC2) [2] uses Xen and Google's Compute Engine [4] utilizes KVM. However, their adoption is still constrained in the HPC context due to inherent VM performance overheads and system dependencies, especially in terms of I/O [5].

On the other hand, lightweight container-based virtualizations, such as Linux-VServer [6] and Linux Containers (LXC) [7] or Docker [8], have attracted considerable attention recently. Containers share the host's resources and provide process isolation, making the container-based solution a more efficient competitor to the traditional hypervisor-based solution. It is anticipated that the container-based solution will continue to grow and influence the direction of virtualized computing.

This paper will demonstrate that container-based virtualization is a powerful technology in HPC environments. This study uses several different container technologies and applications to evaluate the performance overhead, deployment, and isolation techniques for each. Our focus is on traditional HPC use cases and applications; more general use cases have been covered by previous work, such as that by Lucas Chaufournier [17].

This paper is organized as follows: Section II provides an overview of virtualization techniques; Section III presents the different container runtime environments; Section IV describes the motivation behind our study; Section V presents the experiments performed in order to evaluate both application performance and the system overhead incurred in hosting the environment; Section VI presents related work. Conclusions and future work are presented in Section VII.

II. Virtualization

In this section we provide some background on the two types of virtualization technologies that we study in this paper.

A. Hardware Virtualization

Hardware virtualization involves virtualizing the hardware on a server and creating virtual machine instances (VMs) that provide the abstraction of a physical machine (see Figure 1a). Hardware virtualization involves running a hypervisor, also referred to as a virtual machine monitor (VMM), on the bare-metal server. The hypervisor emulates virtual hardware such as the CPU, memory, I/O, and network devices for each virtual machine. Each VM then runs an independent operating system and applications on top of that OS. The hypervisor is also responsible for multiplexing the underlying physical resources across the resident VMs.

Modern hypervisors support multiple strategies for resource allocation and sharing of physical resources. Physical resources may be strictly partitioned (dedicated) to each VM or shared in a best-effort manner. The hypervisor is also responsible for isolation. Isolation among VMs is provided by trapping privileged hardware accesses by the guest operating system and performing these operations in the hypervisor on its behalf.

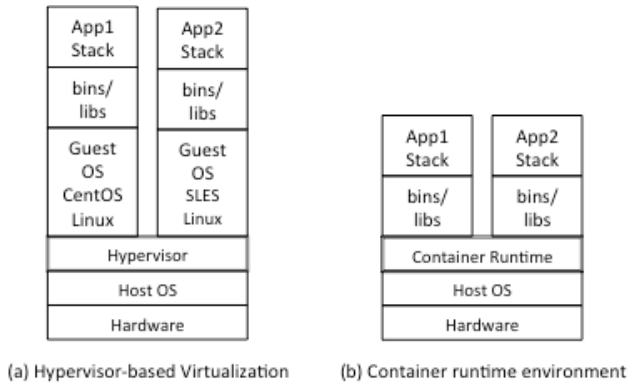


Figure 1. Hypervisor and container-based virtualization

B. *Container-base Virtualization*

Container-based virtualization involves virtualizing the operating system rather than the physical hardware (Figure 1b). OS-level virtualizations are also referred to as *containers*. Each container encapsulates a group of processes that are isolated from other containers or processes on the system. The host OS kernel is responsible for implementing the container environment, isolation, and resources. This infrastructure allocates CPUs, memory, and network I/O to each container.

Containers provide lightweight virtualization since they do not run their own OS kernels like VMs, but instead rely on the underlying host kernel for OS services. In some cases, the underlying OS kernel may emulate a different OS kernel version to processes within a container. This is a feature often used to support backward OS compatibility or to emulate different OS APIs such as in Solaris zones [10].

OS virtualization is not a new technology, and many OS virtualization techniques exist including Solaris Zones, BSD-jails [28], and Linux LXC (Figure 2). The recent emergence of Docker, a container platform similar to LXC but with a layered file system and added software engineering benefits like building and debugging, has renewed interest in container-based virtualization.

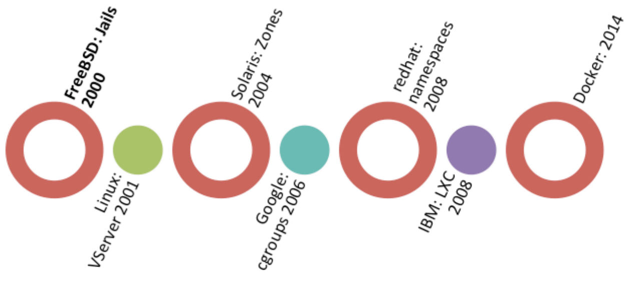


Figure 2. Container technology

Linux containers in particular employ two key features: *control groups* and *namespaces*.

Control groups are a kernel mechanism for controlling the resource allocation to process groups [11]. *Cgroups* exist for each major resource group type: CPU, memory, network, block I/O, and devices. The resource allocation for each of these can be controlled individually, allowing the complete resource limits for a process or a process group to be specified.

Namespaces provide an abstraction for a kernel resource that makes it appear to the container that it has its own private, isolated instance of the resource [12]. In Linux, there are namespaces for isolating process IDs, user IDs, file system mount points, networking interfaces, IPC, and host names.

C. *Containers in HPC*

Implementing existing enterprise container solutions in HPC systems will require modifications to the software stack. HPC systems traditionally already have resource management and job scheduling systems in place, so the container runtime environments will need to integrate into the existing system resource manager. The container infrastructure should follow the core concepts of containers. That is, to abstract the application from the host's software stack, but be able to access host-level resources when necessary, notably the network and storage. Unfortunately, modifying core components of the host to support containers introduces a portability problem. Standard resource managers provide the interfaces required to orchestrate a container runtime environment on the system.

A resource manager such as Moab/Torque on Cray systems utilizes scripts that are the core of the job execution, and these job scripts have the responsibility for configuring the environment and passing the required information at process start. In our experiments we use Cray ALPS as the application launcher to start both parallel MPI applications and serial workloads.

III. CONTAINER RUNTIME ENVIRONMENTS

In this section, we describe the container environments used in our experiments and certain common container attributes pertinent to this study. For the purposes of this paper, we defined container environments to be the tools used to create, manage, and deploy a container. We used two container environments used by the general community: runC [8] and rkt [9]; and two from HPC — Shifter [18] and Singularity [19].

A. *runC*

The command-line utility runC executes applications packaged according to the Open Container Initiative (OCI) format [25] and is a compliant implementation of the Open Container Initiative specification.

The program, runC, integrates well with existing process supervisors to provide a container runtime environment for applications. It can be used with existing resource management tools, and the container will be executed as a child of the supervisor process.

Containers are configured using the notion of *bundles*. A bundle for a container is defined as a directory that includes a specification file named *config.json* and a root file system.

B. rkt

The application container engine rkt is the container manager and execution environment for Linux systems. Designed for security, simplicity, and compatibility within cluster architectures, rkt discovers, verifies, fetches, and executes application containers with pluggable isolation. This container engine can run the same container with varying degrees of protection, from lightweight, OS-level namespace and capability isolation to heavier, VM-level hardware virtualization.

The command-line primary interface is a single executable rather than a daemon process. The command-line utility leverages this design to easily integrate with existing init systems like systemd, as well as with advanced cluster orchestration environments, like SLURM and Kubernetes. rkt implements a modern, open, standard container format, the App Container (appc) [26], but can also execute other container images like those created with Docker.

C. Singularity

Singularity is a lightweight, non-invasive, easily implementable container infrastructure that supports existing workflows and focuses on application portability and mobility.

With Singularity you can build containers based on your host or predefined operating system and define the execution environment. Processes inside the container can be single binaries or a group of binaries, scripts, and data.

D. Shifter

Shifter is a software package that allows user-created images to run at NERSC. These images can be Docker images or other formats. Using Shifter, you can create an image with your desired operating system and easily install your software stacks and dependencies. If you make your image in Docker, it can also be run at any other computing center that is Docker-friendly. Shifter also comes with improvements in performance, especially for shared libraries. Shifter can leverage its volume-mounting capabilities to provide local disk-like functionality and IO performance

E. Container Common Attributes

This section describes the common attributes to expect of all container runtime environments.

1) Host Level Access: The ability to accesses the host-level resources from within the container. Typically these are the host's network interfaces, either TCP/IP or Cray Aries native interface.

2) Privileged Operations: All jobs need to be run by the user and as the user. The system workload managers will enforce this policy. If the container runtime requires root or elevated status, then special system configuration must be applied.

3) Runtime Environment Pass-through: A common workflow on Cray systems allows the user to select different programming environments at runtime via the modules command [20], to either select programming tools or versions of libraries. Typically, this encompasses modifying or adding to the user's environment which is then exported by the batch system to the compute engine. For containers we need to inherit these values to ensure the correct operation.

IV. MOTIVATION

There have already been several studies focused on the performance of general container-based solutions [13-14]. The results indicate that the container-based application can deliver near-native performance for HPC applications. This study looks at different container runtimes, two from open-source and two currently being used in HPC data centers. It also considers how these different container-based solutions can integrate into existing infrastructures and still yield almost native performance.

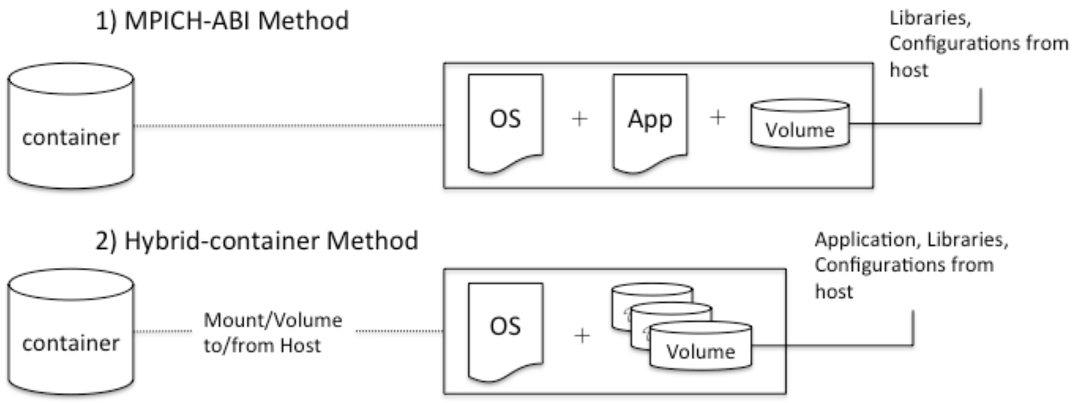
We decided to measure the container start times, application performance, and the overall complexity of integration into an existing infrastructure.

This study motivated us to answer a set of challenging questions: Can we improve the application container performance to deliver near-native performance for the different container runtime environments? What are the fundamental performance barriers when running MPI applications within a container? Can we propose a new design to overcome the bottleneck and significantly improve application performance on such container-based HPC systems?

V. EXPERIMENTS

This section studies the performance and isolation of the different container-based runtime environments. We performed several experiments using a set of containerized MPI and serial applications running on Cray systems. All applications running within containers require host-native access to the Cray Aries network and to a shared file system (Lustre).

Two methods were used to build images for the experiments. Figure 3 illustrates the two different methods used: MPICH-ABI and Hybrid-container builds.



1) MPICH-ABI Method
container
OS
App
Volume
Libraries, Configurations from host
2) Hybrid-container Method
container
Mount/Volume to/from Host
OS
Volume
Application, Libraries, Configurations from host

Figure 3. HPC container models

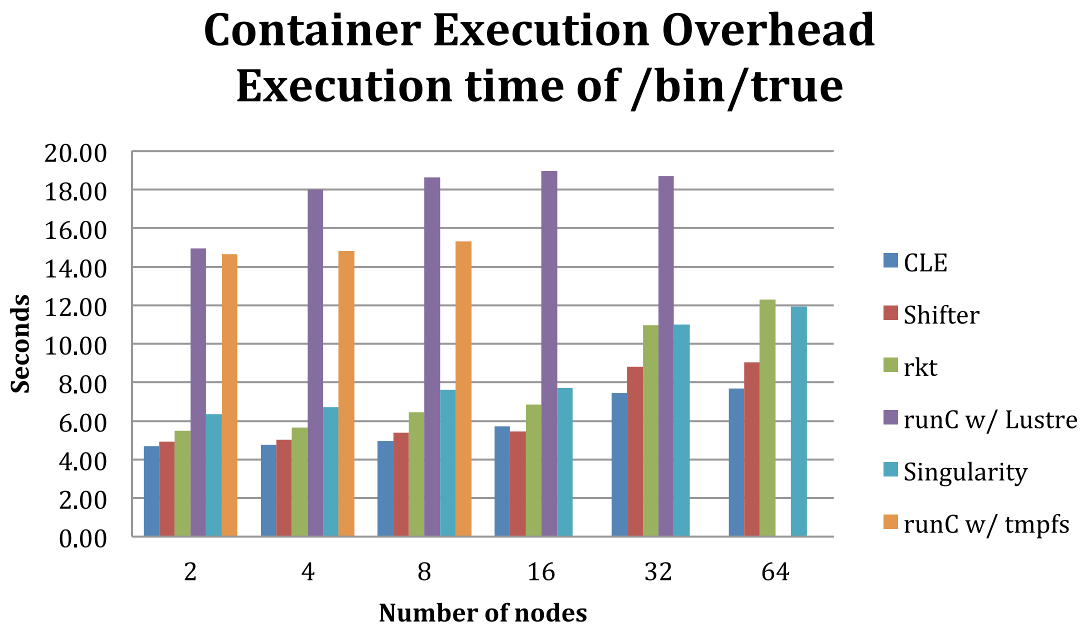


Figure 4. Container execution overhead

1) *MPICH-ABI Method*

MPICH-ABI compatibly defines a standard for interoperability between MPI vendors. An application can compile against one implementation, such as MPICH and execute using another interface such as Cray MPI, which supports the MPICH ABI interface [21]. At runtime the `LD_LIBRARY_PATH` environment variable is set to the Cray MPI version of the MPICH-ABI library and its dependencies in order to replace the MPI libraries in the image.

This method allows an application compiled with MPICH for a TCP network to make use of the Aries network and get native performance. For the experiments done here, the shared libraries were copied to a shared file system (Lustre) and mounted into the container at runtime.

2) *Hybrid-Container Method*

This method takes advantage of using prebuilt applications, using a container to provide isolation for the host environment. This mode of operation requires the container to share the host's file system in order to execute the binary and to read configuration files.

A. *Container Execution Overhead*

To compare container execution overhead, we measured the difference in application startup as compared to standard CLE. We measured how long it takes to start up a simple application as a function of the container runtime environment and at different node counts.

Figure 4 illustrates launching the `/bin/true` binary (essentially a no-op) using different container runtimes at different scales. As the number of nodes increases, both Shifter and Singularity perform well, whereas runC had the largest startup cost due to the fact that a complete image must be copied to the node prior to execution. As with the Docker architecture, each node requires its own image store in which to store the images and container data. rkt also took longer, due in part to having to create a per-node instance of the container in tmpfs. In this case, the actual image data could be pre-fetched, leading to a lower execution time than runC.

In the standard CLE case, we can see a slight increase in execution time as the node count increases. This is to be expected as this system was configured with Resource Utilization Reporting (RUR), which adds some per-node overhead.

The use of runC resulted in node failures when the container was stopped; this was traced to a kernel bug in the umount VFS layer of the kernel. Unfortunately, this prevented further analysis of runC and the rest of the study continued using the three remaining container runtimes.

VI. Performance Overhead on HPC Applications

This section presents an analysis of the performance overhead in container-based systems for HPC applications. For that, we conducted experiments using the NAS Parallel Benchmark (NPB) benchmark suite [23]. NPB is derived from computational fluid dynamic (*CFD*) applications and consists of kernels (*IS, EP, CG, MG, FT*) and three pseudo-applications (*BT, SP, LU*).

The first experiment uses a single node environment to evaluate the performance overhead of the container system. Figure 5 shows the results for each NPB benchmark using the Cray MPI implementation. In all cases, the different container runtimes performed almost identically. Not only did the different container runtimes produce the same results, but when compared to native CLE, the performance was also the same.

When evaluating multinode environments, the influence of the network is the primary metric to be discussed since the network has a direct impact on performance. Figure 6 illustrates performance of the NPB benchmarks on 256 nodes. As with the single node case, the performance of container benchmarks is similar to that of native CLE.

Typically, containers are isolated from the host. One important exception to this rule is when using the Cray Aries network. Getting the best performance for an MPI application running within a container on a Cray system requires access to hugepages [24].

To illustrate the effect of hugepages on application performance, Quantum ESPRESSO [27] was run with and without hugepage support, shown in Figures 7 and 8. When using hugepages, the performance matches native CLE.

VII. Related Work

Several papers have explored container use in HPC environments. As presented by Bahls [13], container HPC applications using the MPICH-ABI method can get good performance on Cray XC40™ systems. Xavier [15] concludes it is possible to get good HPC performance using commodity cluster hardware/software.

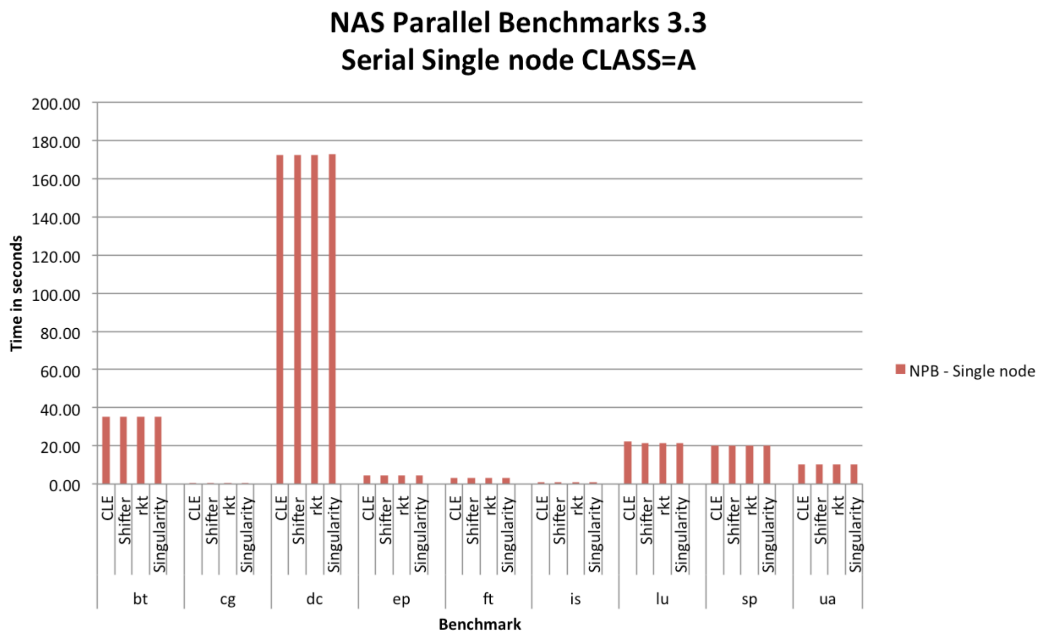


Figure 5. NPB single node results

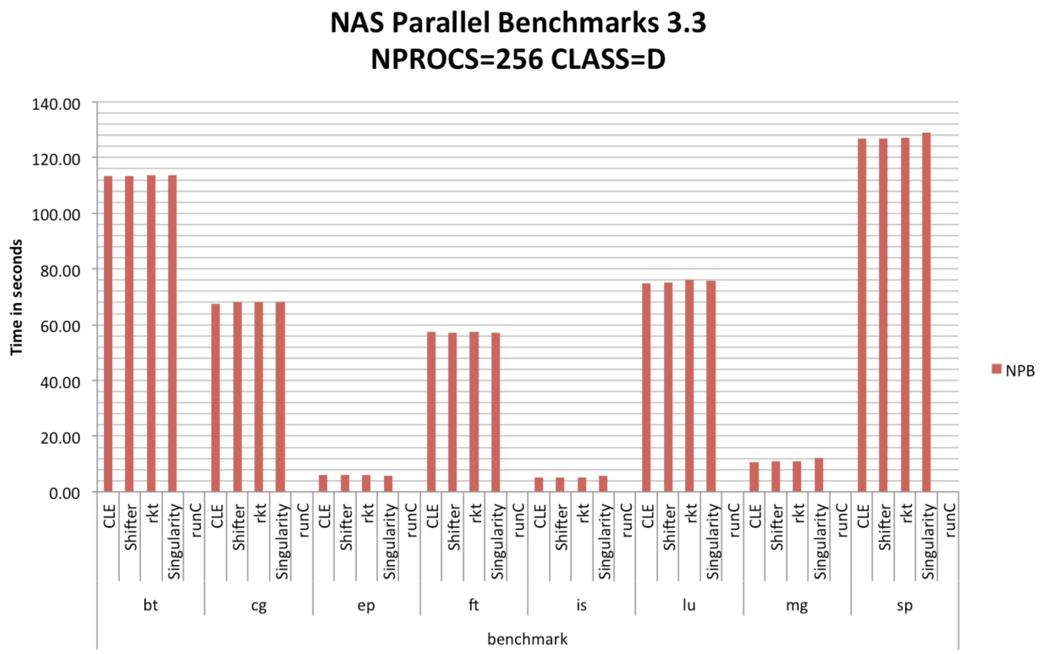


Figure 6. NPB multinode results

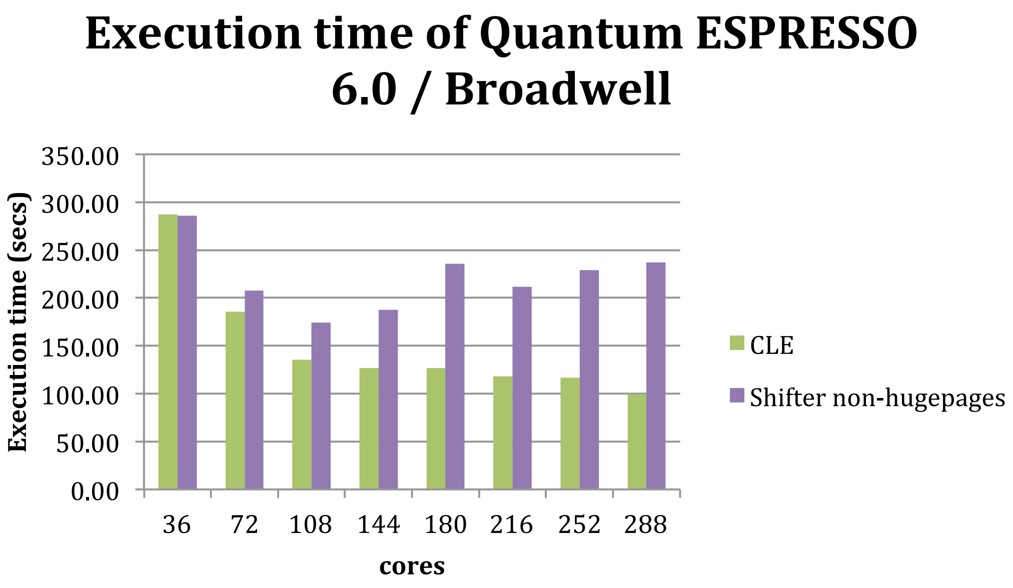


Figure 7. Quantum ESPRESSO without hugepage support

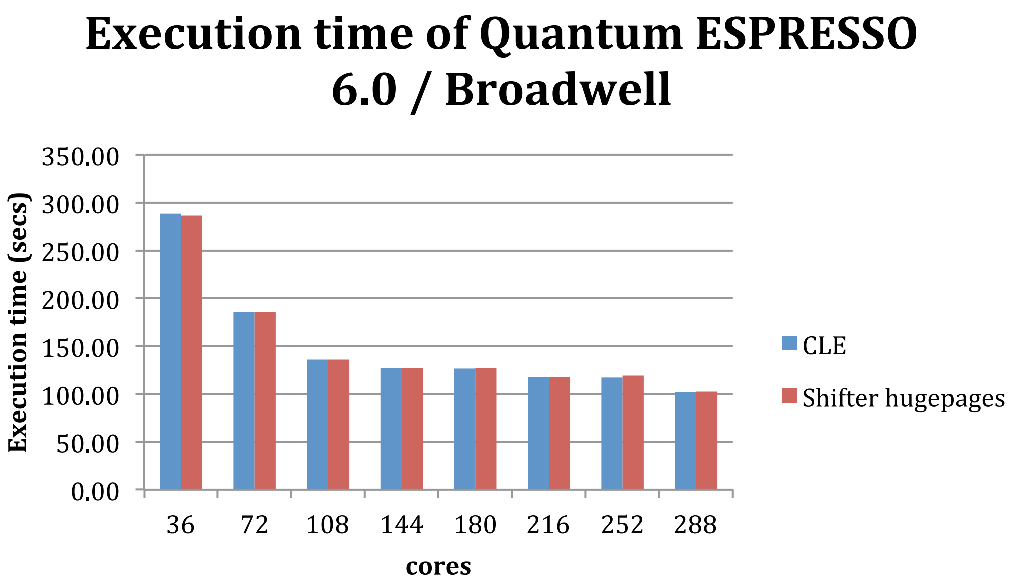


Figure 8. Quantum ESPRESSO with hugepage support

Our results also demonstrate that it's possible to get good container performance if the infrastructure and application can use host-level resources, regardless of the container runtime environment. In our case, using the host network, native MPI libraries, and kernel level interfaces resulted in near-native performance.

As discussed in other research studies using standard clusters [15-16], the major limitation of container technology is its ability to handle I/O intensive applications. By allowing access to host-level resources, we reduce this bottleneck.

VIII. CONCLUSION AND FUTURE

Container-based computing is a new technology that is being adopted at a remarkable rate in the enterprise data center. The flexibility and productivity gains are revolutionary for software development and deployment. While this technology is still in development in HPC, customers are now requesting the ability to execute containers at scale on HPC systems.

During this investigation, we showed the performance characteristics and demonstrated that we can achieve almost native performance if the environment is properly configured. While each of the runtimes delivered nearly the same performance, actual startup costs varied wildly depending on the container environment and how the image was staged.

Each container runtime environment presented a different set of configuration parameters: HPC variants employed a simpler set of parameters and a deployment mode, whereas the open source variants have a comprehensive set of configuration and runtime options.

For future work, we plan to investigate a common workload manager interface to different container runtime environments and provide an alternate API to orchestrate containers as well as a batch interface.

Acknowledgment

The author would like to acknowledge the following people for their assistance and insights during this research: Stephen Behling, Pierre Carrier, Steve Warren, Kim McMahon, Sung-Eun Choi, and Brad Chamberlain from Cray, and Douglas Jacobsen and Richard Shane Canon from NERSC, Lawrence Berkeley National Laboratory.

References

[1] Xen, http://www.xen.org

[2] Kernel-based Virtual Machine (KVM), http://www.linux-kvm.org

[3] Amazon EC2 http://aws.amazon.com/ec2

[4] Google Compute Engine (GCE), https://cloud.google.com/cpompute

[5] J.Liu. "Evaluating Standard-Based Self-Virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV Support", in Proceeding of 2010 IEEE International Symposium Parallel & Distributed Processing (IPDPS). IEEE, 2010, pp. 1-12

[6] Linux virtualization for GNU/Linux systems, http://linux-vserver.org

[7] LXC (Linux Containers), https://linuxcontainers.org

[8] Docker container platform, https://www.docker.com

[9] CoreOS container infrastructure, https://coreos.com

[10] Oracle solaris Zones https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm#OPCUG426

[11] Linux Cgroups, https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[12] Linux Namespaces, http://man7.org/linux/man-pages/man7/namespaces.7.html

[13] Donald Bahls, "Evaluating Shifter for HPC applications", presented at the Cray User Group., London, UK., 2016

[14] D. Jacobsen, S. Canon, "Contain this, unleashing Docker for HPC," presented at the Cray User Group., Chicago, IL., 2015

[15] Miguel Gomes Xavier, Marcelo Veiga Neves, Fabio Diniz Rossi, Tiago C. Ferreto, Timoteo Lange, Cesar A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments", Pontifical Catholic University of Rio Grande do Sul (PUCRS) Porto Alegre, Brazil., 2013

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

[17] Lucas Chaufournier, Prateek Sharma, Prashant Shenoy, Y.C. Yay, "Containers and virtual machines at scale: a comparative study", Middleware '16 Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016

[18] Shifter, https://www.nersc.gov/research-and-development/user-defined-images

[19] Singularity, http://singularity.lbl.gov

[20] Modules - Software Environment Management, http://modules.sourceforge.net

[21] MPICH ABI Compatibility Initiative, https://www.mpich.org/abi/

[22] runC, https://github.com/opencontainers/runc

[23] NAS Parallel Benchmarks, https://www.nas.nasa.gov/publications/npb.html

[24] Huge Pages – The Linux Kernel Archives, https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[25] Open Container Initiative, https://www.opencontainers.org

[26] App Container Specification, https://github.com/appc/spec/blob/master/SPEC.md

[27] Quantum ESPRESSO, http://www.quantum-espresso.org

[28] BSD-Jails, https://www.freebsd.org/doc/handbook/jails.html