Containers in Research: Initial Experiences with Lightweight Infrastructure

Spencer Julian
Purdue University
155 S. Grant St.
West Lafayette, IN
smjulian@purdue.edu

Michael Shuey
Purdue University
155 S. Grant St.
West Lafayette, IN
shuey@purdue.edu

Seth Cook
Purdue University
155 S. Grant St.
West Lafayette, IN
cook71@purdue.edu

1. ABSTRACT

HPC environments have traditionally existed installed directly on hardware or through virtual machine environments. Linux Containers, and Docker specifically, have gained extensive popularity; we believe this current trend toward containers and microservices can be applied to HPC to improve efficiency and quality of development and deployment. User interest in Docker is rising, with several communities planning production deployments. We describe some of our site's experiences, along with an autoscaling web cluster and an autoscaling PBS-based computational cluster we have developed that are currently in a pre-production testing phase. Some basic performance tests are covered, comparing network and filesystem performance between a native Docker environment and a traditional Red Hat-based environment. In our tests, we noticed negligible differences in computational performance when run out of the box, approximately 0.4%, but we required some minor tweaking in the form of additional docker plugins to achieve similar or better performance in the network and filesystem tests. While additional testing is needed for some aspects of computational clusters, particularly RDMA performance, we believe initial testing indicates Docker containers are ready for broader adoption at larger-scale production environments.

CCS Concepts

•**Software and its engineering → System administration;** *Software configuration management and version control systems; Maintaining software;* Software libraries and repositories;

Keywords

Docker; container; LXC; cgroups

2. INTRODUCTION AND BACKGROUND

In the 1930s, international shipping companies transitioned away from moving individual pieces of cargo onboard ocean-going ships, and adopted a standard size of pre-fabricated cargo container. This conversion allowed shipping companies to streamline operations in ports, reducing congestion and ultimately reducing both shipping time and losses due to mishandled cargo. [1] Modern software containers are named for this cargo shipping phenomenon; by adopting a similar set of standard form-factors for software deployment, proponents hope to increase overall organizational agility by reducing the time and complexity of software deployment, without sacrificing performance.

Current container technology has its roots in the `cgroups` feature in the Linux kernel. `cgroups` is a nickname for process control groups – a mechanism for isolating, limiting, and accounting for resource usage across a collection of processes. [2]. This core functionality has given rise to low-level management library suites like LXC (LinuX Containers), and eventually higher-level frameworks such as Docker or Rkt. These frameworks aim to simplify deployment for the end user (or software developer), and abstract away many of the complexities involved in developing and deploying "containerized" software environments.

Containers usually encompass all software necessary to run a single application – forming a "microservice", an environment dedicated to a single task. This approach lends itself to a high degree of flexibility; the container environment can provide different tools from the host operating system, or a different Linux distribution entirely. Small microservices can also be deployed quickly (as no unnecessary software or configuration must be provisioned), allowing containers to be easily created or destroyed to keep up with changes in application demand, or re-deployed to work around failures in a large datacenter. Such easy-to-deploy environments may also help enable continuous integration of changes. Proposed changes, feature additions, or bug fixes can first be tested in a private container in an automated fashion, before being pushed to a production environment – allowing software changes to be deployed continually, with little risk to production operations.

These benefits have driven considerable interest in frameworks like Docker for business and cloud market segments – but high-performance or scientific computing environments have focused primarily on custom-built or heavily modified solutions, and are only starting to begin widespread adoption of containers. Hardware requirements for both simulation and data analysis may be converging to a common subset [3, 4], with different software tools used for different workflows. We believe that the ability to more flexibly deploy software environments (as afforded by container frame-

XSEDE16, July 17 - 21, 2016, ,

ISBN 978-1-4503-4755-6/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2949550.2949562

works like Docker) will be a key benefit. This deployment model is slowly being adopted by developers of research systems; Docker containers for popular scientific gateways like Galaxy or JupyterHub can already be found on Docker's Hub system [5], and it seems likely more will emerge over time.

In the following section, we present our initial use cases for Docker's container technology. Starting with simple web server containers, we show the productivity benefits available as user communities embrace this approach, and then discuss our initial efforts at building containers able to emulate (or potentially, replace) a mid-scale high-performance compute (HPC) cluster.

3. USE CASES

Research computing, as a topic, has broadened considerably in recent years. Originally the realm of high-end HPC users and scientific simulations, research computing now encompasses a wide variety of workflows – including scientific gateways from an increasingly diverse customer base. [6] Many of these new communities have developed custom software for their gateways that may not be compatible with a software stack in use in a predominantly HPC-oriented environment. That was the case at our site; requests for new scientific gateways with requirements outside our usual supported environments became our first use cases for Docker-based containers.

3.1 Initial Web Environments

A research group at our site needed to deploy new gateway tools developed under the latest version of Debian Linux, and requiring a very recent version of PHP. Debian Linux is unsupported at this site, as was the specific version of PHP; initial estimates indicated 1-2 weeks to deploy a service to meet the group's needs. Instead, we elected to install Docker on top of the group's existing software environment, and deploy an off-the-shelf PHP container, provided by the Docker Hub [5]. This cut the deployment time down to a single afternoon, greatly improving the group's overall service delivery time and lessening the burden on the site's system administration team.

This same approach was used weeks later, to deliver a JupyterHub lab notebook server. The JupyterHub team produced a Docker container [7] as a reference build of their solution. Again, the deployment time of a new service was reduced from days to hours, as the reference container was easily deployed on top of Docker in an existing support environment. We believe this deployment scenario will become increasingly common, as more scientific gateway and research tool developers produce more Docker-based reference implementations of their projects. We are already experimenting with containerized deployments of tools such as Galaxy and Apache Hue, and see a panoply of container use cases emerging for Hadoop and the "big data" space. [8, 9, 10, 11]

3.2 Customer-Driven Docker Environments

After two months of operation, our first Docker-enabled research group needed to expand. Their new gateway environment required more components – four different research tools, an administrative system, and a dedicated database. Many of the research tools were in continuous development, which would normally require additional access privileges for the software developers and involvement of a local system administrator. Since this team had piloted our initial use of Docker, we opted to try a broader roll out.

All six software environments were deployed as Docker containers, with key configuration mapped as a Docker storage volume from existing project file servers. As before,the use of off-the-shelf containers greatly accelerated initial deployment. In this case, though, project developers were also provided with direct access to the containers' configuration files, as well as an on-site `git` repository to host their work. The underlying Docker host systems have been configured to allow developers to build, test, and deploy their containers, so development can continue without the involvement of site administrators. In fact, the only time site administrators need to intervene is to restore from catastrophic failure (restoring from system-level backups) or to provision additional host resources. Again, Docker's modularity and isolation helps improve our service delivery times, and leads to better outcomes for our research teams.

3.3 Current Development: Autoscaling

Off-the-shelf Docker containers are beginning to help our site's ability to more rapidly deliver services, but requests from some of our larger communities may need more resources than a single container (or even a single server) can provide. In these cases, we need to be able to scale – a scenario where Docker's lightweight deployment capabilities are definitely an asset. We've begun developing two prototype systems, one each for generic web and compute environments, to enable workloads to be automatically scaled up and down as load demands.

In both prototypes, new container requests are sent to Docker Swarm. Swarm is a (relatively new) meta-scheduler[1] for Docker; it will launch containers on any Docker host available. Assuming an appropriate load balancer or task scheduler, this can spread a workload across a large number of physical hosts (improving both system capacity and, in many scenarios, resiliency).

3.3.1 Web Clusters

We have built a prototype system for autoscaling a simple web cluster, based on containerized web servers. In this environment, a pair of HAProxy containers, configured as a highly-available pair, are used to spread requests to one or more back-end NGINX web severs. The HAProxy containers monitor response times for web requests using a simple Python script; if request times drop below a preconfigured minimum, additional NGINX containers are dynamically created and configured as HAProxy backends to help meet the assumed request demand. Similarly, as the response time decreases, NGINX containers will be slowly eliminated until either request time begins rising again or a configured minimum number of back-end NGINX containers is reached. New container startup times are generally in the 10-20 second range, though that will vary based on the load of the underlying Docker host systems.

Our prototype here assumes a relatively simple application stack – HAProxy servers connect to NGINX containers via a private overlay network, with simple static web pages.

[1]The implications of a container-level task scheduler are an active area of research. We chose Swarm based on its ease of implementation, though Mesos or Kubernetes are also strong candidates in this space.

However, this basic design should be extremely scalable; a database container can easily be added to the overlay network, and NGINX containers can be replaced with more complex software systems. We expect to use this prototype to automatically scale a wide variety of web-enabled research tools.

3.3.2 HPC Clusters

The majority of research groups at our site use some sort of clustered compute solution. For most, a general-purpose HPC cluster (A Linux-based OS, a variety of compilers and MPI flavors, a batch scheduler, etc.) will suffice – but we are beginning to see a need for different software stacks. A certain project may require specialized tools (e.g., the Open Science Grid, and related high-energy physics experiments) or radically different software environments like Hadoop or Spark. Ideally, we would like to deploy a single set of capable cluster hardware, and dynamically change the operating environment as jobs present themselves in each of several custom software environments. This has led to the development of our second prototype system: a dynamically-scaling batch-scheduled compute cluster.

For our testing, we built an environment with CentOS 7 and Adaptive Computing's Moab scheduler. Moab supports an elastic computing environment, to run external programs to deploy additional nodes when particular metric thresholds are reached. Originally designed for systems like OpenStack or Amazon's AWS, we adapted this functionality to launch new Docker containers on demand.

The prototype centers around three containers: a user-accessible front-end system (where the container runs `sshd`, to accept user logins), the scheduler (which runs both the resource manager and the Moab scheduler), and a compute node container (which launches a `pbs_mom` daemon at start). All three containers integrate with our site's existing LDAP authentication framework, and share several of our production filesystems for `/home` and project storage. Initial testing was conducted on older hardware, with a 10 Gbps Ethernet interconnect network.

From a user's perspective, this environment is nearly indistinguishable from our traditional cluster computing environment. At launch, only a single node is available, and all batch queues are empty. As jobs are submitted, Moab examines the pool of available nodes. If the cluster lacks sufficient nodes to run a job, and no queue restrictions prevent it from starting, Moab will launch additional containers until the job can be started. This process is dependent on the size of the compute node container (which can be surprisingly large, as it may contain a large number of libraries and tools), but usually occurs in well under a minute – even on relatively modest older hardware. As jobs exit, Moab will automatically destroy unused containers via external Python scripts, until more jobs are presented.

While still a prototype, this system offers an intriguing possibility for future HPC deployments. If clusters automatically resize themselves, based on utilization, we expect to be able to share the same underlying hardware with a multitude of users – potentially with very different software requirements.

4. PERFORMANCE

As others have noted [12], containers show a negligible impact on performance when compared to running on the native operating system. As we're considering use cases for Docker in an HPC environment, however, we have worked to quantify these differences at our site.

Our experiments were run on older 24-core AMD Opteron systems, with 2.1 GHz processors. All systems have 48 GB of memory, and a 10 Gbps Ethernet connection (used for `iperf` tests, below, as well as a link to all network storage systems). While our site makes heavy use of Red Hat and CentOS Linux for production systems, we have been piloting the use of the minimalistic Linux platform CoreOS for our more recent container testing. Systems were installed with RHEL 6.5, for native tests, and booted to CoreOS 899.5.0 for use with Docker. CoreOS 899.5.0 features Linux kernel version 4.3.3 and Docker version 1.9.1.

4.1 HPL

Our LINPACK tests were confined to a single core intentionally, to illustrate the processing overhead inherent in a Docker container. We also wanted to avoid scaling `xHPL` across multiple nodes, as we do not yet have an RDMA-capable interconnect in our test environment (as discussed in more detail in Section 5).

HPL performance was relatively consistent between the RHEL 6.5 and Docker environments, as shown in Figure 1. The native RHEL 6.5 system showed an average performance of 7.839 GFLOPS, versus 7.811 GFLOPS in a containerized environment (approximately a 0.4% slowdown). Interestingly, the native RHEL 6.5 environment showed a higher deviation than the Docker environment (a standard deviation of 0.032 versus 0.001) – though this variation is most likely due to the kernel differences between RHEL 6.5 (Linux 2.6.18) and CoreOS 899.5.0 (Linux 4.3.3).

4.2 Network Throughput

Our network throughput was measured with the `iperf` tool. Two measurements were taken – upload bandwidth, followed by download. In both cases, the remote endpoint was a similarly-configured server running a native RHEL 6.5 operating system. As shown in Figure 2, upload bandwidth (on this hardware) is slightly lower than download bandwidth.

Initially, our Docker environment performed extremely poorly – but only during upload tests. Further investigation revealed that Docker will create a host-based NAT (Network Address Translation) configuration by default; all outgoing packets from our test machine were being inspected and modified by the kernel. We switched to a "bridge" driver created using the pipework project [13], as we suspected our relatively old test hardware would be unable to conduct packet modification at these speeds. The bridge driver dynamically configures a network bridge between the container processes and the host network interface, then uses Linux's "macvlan" capabilities [14, 15] to provide extremely lightweight isolation from the host system. This provided greatly improved performance (see Figure 2); final upload throughput was 8.43 Gbps under Docker (versus 8.26 Gbps in RHEL 6.5), with a download throughput of 9.37 Gbps (versus 9.38 Gbps in RHEL 6.5). We believe these slight discrepancies to be mostly due to kernel differences between RHEL 6.5 and CoreOS 899.5.0, as mentioned in Section 4.1.

4.3 Network File System throughput

Storage throughput was tested to an NFSv3 server, backed

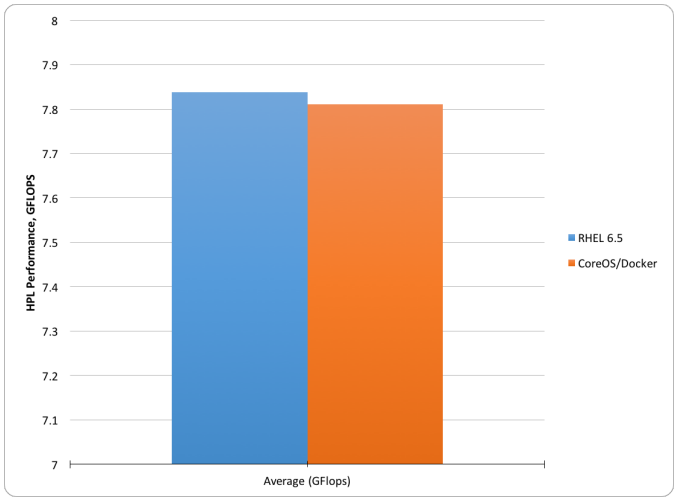

Figure 1: Single-core HPL performance

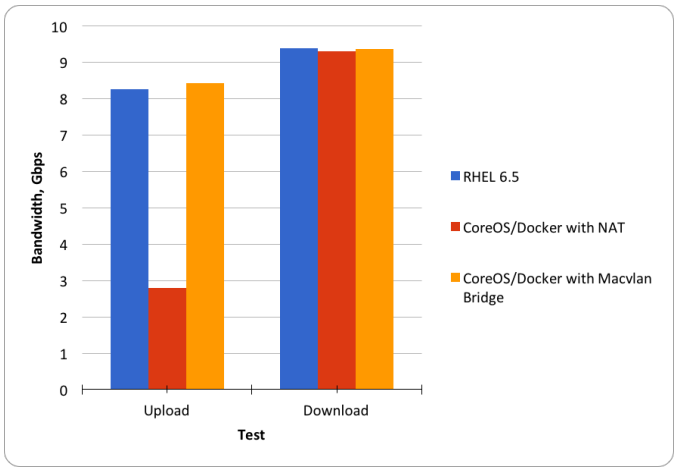

Figure 2: Network throughput, as measured by `iperf`

by a very capable GPFS file system. All measurements were made with the `iozone` tool, working on a 60 GB file to guarantee we exceed the memory capacity of our 48 GB test system, and with a record size of 256 kB. Basic streaming read and write tests (and re-read and re-write tests), as well as random read and write test results, are shown in Figure 3. For containerized tests, a volume was mapped into the container (and used to store `iozone`'s data files).

The "RHEL 6.5" curve in Figure 3 provides a baseline for performance. The second curve, "CoreOS Native", provides a similar baseline for the newer kernel version in CoreOS 899.5.0 (to avoid some of the possible sources of variation in the results from sections 4.1 and 4.2). Initial Docker runs, however, were considerably slower than either RHEL 6.5 or native CoreOS.

By default, Docker data volumes are a simple directory mapping – taking the named directory in the host OS and making it appear at the named location within the container. Behind the scenes, Docker uses OverlayFS, a relatively new filesystem to provide layered filesystem functionality. Obviously, this comes at a marked performance penalty ("CoreOS/Docker with OverlayFS" curve, Figure 3).

Docker 1.9 supports a plug-in API, to extend its data volume capabilities. After some searching, we discovered an NFS volume plugin [16] that would directly mount an NFS server into a container's filesystem namespace as a data volume. These results are shown in the "CoreOS/Docker with NFS Plugin" curve in Figure 3, and are much more in line with the expected performance of our system. Surprisingly, this configuration also proved the fastest for random write IO; further investigation is required to determine the root cause behind this improvement.

5. FUTURE WORK

We see a need for improved data volume plug-ins (as evidenced by results in Figure 3). Unfortunately, traditional HPC-centric parallel file systems (e.g., Lustre, GPFS) do not lend themselves to dynamic mounting and unmounting to ephemeral containers; it's likely these sorts of storage systems will need to be mounted to the underlying host, then mapped into containers as data volumes. We are currently developing improvements to the plug-in used in our testing in Section 4.3, to allow for higher throughput to these sorts

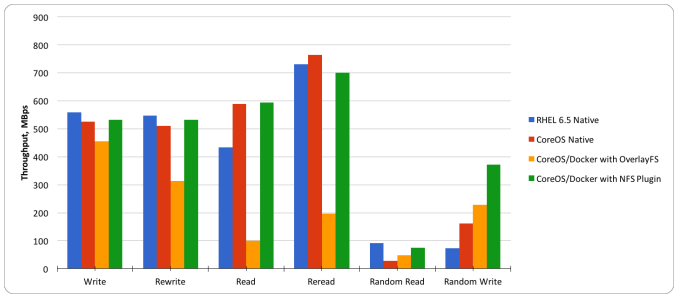

Figure 3: File server throughput, as measured by `iozone`

of filesystems.

We also recognize the lack of an RDMA interconnect in our present test facilities. Based on earlier work by Mellanox [17], we believe it will be possible to get near-hardware levels of performance from modern RoCE and Infiniband interconnects. Our initial experimentation here appears nominally successful, and we are planning a more thorough test as soon as upgrades to our experimental facility are complete.

Finally, we see sufficient research benefit from containers, and from Docker in particular, to begin building a production-ready pilot system. Our initial phase will be based on CoreOS, to rapidly provision hardware to serve a Docker Swarm. Machines will be dispersed across three locations at our site, providing a measure of fault-resilience and a means to test it. A similarly distributed storage system will be required, to provide similar levels of fault-tolerance to persistent data. We have initially selected Ceph [18] to provide basic block and filesystem storage, and are completing the plans for a small-scale trial installation as of this writing. If successful, this environment should provide strong guarantees of availability for scientific gateways and web-deployed research tools, as well as a test bed for future container investigations.

6. RELATED WORK

Jetstream at Indiana University [19] and Bridges at the Pittsburgh Supercomputing Center [20] both offer similar services to science gateways and non-traditional workflows, though through an OpenStack-based virtualization system. Comet at the San Diego Supercomputer Center [21] offers very high-performance virtual environments as well, through an in-house system based on Linux KVM. These software defined research environments are excellent demonstrations of the technology, and can be extended to support container systems such as Docker should the need arise.

NERSC has developed Shifter [22], a unique virtualization environment that can convert Docker and other virtualization images into its own environment for execution on NERSC's Cray supercomputers. This is a novel approach which allows for some interoperability between environments, but runs container environments as virtualized jobs on the supercomputer. LBNL has developed Singularity [23] as an alternative; this system is designed to integrate well with HPC systems, and uses Linux container technology to ensure a low-overhead software environment. Singularity uses a custom container format, though, and cannot directly benefit from advances in the Docker community, including available Docker container libraries.

Docker investigations are underway at several research institutions world-wide. We are aware of cluster experiments [24], production load for scientific experiments [25], and work with web-enabled research tools [26] at multiple sites. HPC tool vendors have also begun integrating native support for Docker; for example, IBM has added Docker container integration in Platform LSF, to run containers on an HPC cluster. [27] This allows containers to be executed on an LSF-managed cluster like a conventional job, but with a fully self-contained environment. The xCAT cluster provisioning suite has also added a measure of Docker integration; the latest versions now support using containers to manage xCAT clusters, and master new xCAT images. [28]

7. CONCLUSIONS

Our initial testing shows great promise for Docker in scientific research. While more testing is required for certain aspects of HPC sites (particularly around RDMA interfaces), Docker is ready for many workloads today. Scaling and performance are extremely active areas for the Docker community. Our work shows improvement over earlier testing [12], and coupled with community initiatives toward enhanced RDMA support [13, 17] and data volume management [16], we are confident the last barriers to large-scale HPC adoption are disappearing.

Lightweight containers are an extremely active area of research, and viewed by many [23, 22, 24, 25] as an important mechanism to enable highly customized workflows or software environments. We highlight the nominal performance penalties of this approach for small-scale use cases, and show a potential avenue for supporting both custom container-based environments as well as conventional HPC work in a unified platform, much like the web or cloud environments in other market sectors.

8. REFERENCES

[1] Matthew Heins. *The Globalization of American Infrastructure: The Shipping Container and Freight Transportation*. Routledge, 2016.

[2] Paul B Menage. Adding generic process containers to the linux kernel. In *Proceedings of the Linux Symposium*, volume 2, pages 45–57. Citeseer, 2007.

[3] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve. Big data, simulations and hpc convergence.

[4] Tiffany Trader. Toward a converged exascale-big data software stack. 2016.

[5] http://hub.docker.com.

[6] Alan B. Craig. Science gateways for humanities, arts, and social science. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 18:1–18:3, New York, NY, USA, 2015. ACM.

[7] https://github.com/jupyterhub/jupyterhub.

[8] Nicholas Davis. Exploring adverse drug effect data with apache spark, hadoop, and docker. 2015.

[9] P China Venkanna Varma, KV Kalyan Chakravarthy, V Valli Kumari, and S Viswanadha Raju. Analysis of network io performance in hadoop cluster environments based on docker containers. In *Proceedings of Fifth International Conference on Soft Computing for Problem Solving*, pages 227–237. Springer, 2016.

[10] Rui Zhang, Min Li, and Dean Hildebrand. Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 365–368. IEEE, 2015.

[11] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In *Open Systems (ICOS), 2015 IEEE Confernece on [sic]*, pages 130–135. IEEE, 2015.

[12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172, March 2015.

[13] https://github.com/jpetazzo/pipework.

[14] Victor Marmol, Rohit Jnagal, and Tim Hockin. Networking in containers and container clusters. In *Proceedings of netdev 0.1*, February 2015.

[15] Daniel Lezcano. *LXC.CONTAINER.CONF(5) Man Page*, May 2016.

[16] https://github.com/ContainX/docker-volume-netshare.

[17] Howto create docker container enabled with roce. https://community.mellanox.com/docs/DOC-1506, October 2014.

[18] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[19] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. Jetstream: a self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 29. ACM, 2015.

[20] Nick Nystrom. Introduction to bridges: Connecting researchers, data and hpc. https://www.youtube.com/watch?v=hn3tPkZaY4U, January 2015.

[21] Richard L Moore, Chaitan Baru, Diane Baxter, Geoffrey C Fox, Amit Majumdar, Phillip Papadopoulos, Wayne Pfeiffer, Robert S Sinkovits, Shawn Strande, Mahidhar Tatineni, et al. Gateways to discovery: Cyberinfrastructure for the long tail of science. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, page 39. ACM, 2014.

[22] Douglas M Jacobsen and Richard Shane Canon. Contain this, unleashing docker for hpc.

[23] http://singularity.lbl.gov.

[24] Hsi-En Yu and Weicheng Huang. Building a virtual hpc cluster with auto scaling by the docker. *arXiv preprint arXiv:1509.08231*, 2015.

[25] E Mazzoni, S Arezzini, T Boccali, A Ciampa, S Coscetti, and D Bonacorsi. Docker experience at infn-pisa grid data center. In *Journal of Physics: Conference Series*, volume 664, page 022029. IOP Publishing, 2015.

[26] J Gomes, J Pina, G Borges, J Martins, N Dias, H Gomes, and C Manuel. Exploring containers for scientific computing. In *8th Iberian Grid Infrastructure Conference Proceedings*, page 27.

[27] Bill McMillan and Chong Chen. High performance docking. Technical report, IBM, 2014.

[28] http://xcat-docs.readthedocs.io/en/stable/advanced/docker/index.html.