Evaluation of Docker Containers for Scientific Workloads in the Cloud

Pankaj Saha
State University of New York (SUNY) at Binghamton
Binghamton, New York
psaha4@binghamton.edu

Angel Beltre
State University of New York (SUNY) at Binghamton
Binghamton, New York
abeltre1@binghamton.edu

Piotr Uminski
Intel Technology Poland
Gdansk, Poland
piotr.uminski@intel.com

Madhusudhan Govindaraju
State University of New York (SUNY) at Binghamton
Binghamton, New York
mgovinda@binghamton.edu

ABSTRACT

The HPC community is actively researching and evaluating tools to support execution of scientific applications in cloud-based environments. Among the various technologies, containers have recently gained importance as they have significantly better performance compared to full-scale virtualization, support for microservices and DevOps, and work seamlessly with workflow and orchestration tools. Docker is currently the leader in containerization technology because it offers low overhead, flexibility, portability of applications, and reproducibility. Singularity is another container solution that is of interest as it is designed specifically for scientific applications. It is important to conduct performance and feature analysis of the container technologies to understand their applicability for each application and target execution environment.

This paper presents a (1) performance evaluation of Docker and Singularity on bare metal nodes in the Chameleon cloud (2) mechanism by which Docker containers can be mapped with InfiniBand hardware with RDMA communication and (3) analysis of mapping elements of parallel workloads to the containers for optimal resource management with container-ready orchestration tools. Our experiments are targeted toward application developers so that they can make informed decisions on choosing the container technologies and approaches that are suitable for their HPC workloads on cloud infrastructure. Our performance analysis shows that scientific workloads for both Docker and Singularity based containers can achieve near-native performance.

Singularity is designed specifically for HPC workloads. However, Docker still has advantages over Singularity for use in clouds as it provides overlay networking and an intuitive way to run MPI applications with one container per rank for fine-grained resources allocation. Both Docker and Singularity make it possible to directly use the underlying network fabric from the containers for coarse-grained resource allocation.

PEARC '18, July 22–26, 2018, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-6446-1/18/07…$15.00
https://doi.org/10.1145/3219104.3229280

CCS CONCEPTS

• **Hardware** → **Networking hardware**; • **Software and its engineering** → *Application specific development environments*; • **General and reference** → Performance;

KEYWORDS

Docker, Singularity, scientific workloads

ACM Reference Format:
Pankaj Saha, Angel Beltre, Piotr Uminski, and Madhusudhan Govindaraju. 2018. Evaluation of Docker Containers for Scientific Workloads in the Cloud. In *Proceedings of Practice and Experience in Advanced Research Computing (PEARC '18).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3219104.3229280

1 INTRODUCTION

Containerization technology has gained significant traction in recent years. Containers are the appropriate tool for software development landscape that has adopted microservices and DevOps. Containers have several well known benefits for use in cloud environments: (1) support for a uniform environment for testing and deploying applications; (2) seamless and continuous updates of microservices; and (3) agility to support different languages and deployment platforms.

When containers were initially adopted for micro-services based architectures in the industry, they did not attract the attention of HPC community wherein the focus is primarily on MPI applications for large parallel tasks. Containers were also initially shunned due to reports of possible root escalation vulnerabilities. However, the growing list of features along with the potential to provide high performance, along with portability and reproducibility from development to production environment, has made it critical to evaluate current container technologies for HPC applications in the cloud.

Docker [5] containers are known as lightweight Virtual Machines, which provide networking capabilities and dedicated computational resources. Singularity containers are designed to efficiently in conventional HPC environments. While Slurm [11] is the most widely used scheduler for HPC applications, the industry has successfully used open source technologies in the cloud such as Apache Mesos [3] and YARN [10] for resource management, and container orchestrators like Kubernates [9] and Docker Swarm [6].

As there is support for Docker with these technologies, containerized HPC applications can also avail features such as container migration, resource fairness, and fault tolerance.

Singularity is designed to use the underlying HPC runtime environment for executing MPI applications, whereas Docker is designed to isolate the runtime environment from the host. Also, Singularity focuses on coarse-grained resource allocation whereas Docker can take advantage of the fine-grained allocation of resources per rank.

HPC centers and academic clusters currently do not widely support Docker due to reports of security concerns that root escalation is possible. However, this vulnerability is not a concern in cloud allocations wherein users have root privileges to run their applications and other security modules provide separation between different allocations. As containers keep expanding support for running HPC tasks in the cloud, it is critical to quantify the impact on performance, support for orchestration and scheduling/placement of containers on the resources.

The features of interest to the HPC community, which we present in this paper, include (1) evaluation of support for Infiniband and RDMA communication across MPI ranks; (2) mapping of sshd ports between the container and host machine; (3) determining overhead of containers compared to bare-metal access for memory, cpu, and communication intensive tasks; and (4) study use of overlay networks by container orchestration tools and its effect on HPC applications.

Our experiments were conducted on an NSF funded academic Chameleon cloud [4]. It provides a utility called "Complex Appliances," which offers accessible cluster configurations for the acquired nodes. One of the appliances is called "MPI bare-metal cluster," which comes with "MPICH2" library on CentOS 7 based image. Chameleon Cloud provides InfiniBand supported bare metal nodes.

- We mapped InfiniBand devices of the host machine to Docker containers for RDMA communication across MPI ranks.
- We quantified the performance overhead of MPI applications when run in containers and compared with their bare metal performance. For this evaluation, we used both the host and Docker's overlay network to execute the MPI benchmarks. With Docker, we orchestrated containers using different approaches:
  (1) We ran the container's sshd on a non-standard port and mapped the port with host machines. One container per host was used in this approach such that the container and host node use the same IP address, but run sshd on different ports.
  (2) We used an overlay network, provided by Docker Swarm, which spans across multiple host nodes. We created a *one container per node* setup and plugged all of them to the same overlay network. This network assigned separate IP addresses to all containers under the same subnet address.
  (3) We used a similar setup as approach (2), but used multiple containers per host node attached to the common overlay network.
- We used different classes (CPU , memory, and latency sensitive) of MPI applications for benchmarking. We observed for fixed number of MPI ranks how the performance varies when the number of nodes changes in the cluster. We also compared communication with InfiniBand vs Ethernet for the different classes of applications.

2 BACKGROUND

Linux container (LXC) provides a virtual layer on top of the Linux host Operating System (OS) allowing multiple Linux systems to run in isolation. Containers separate the runtime environment from the underlying host resources and networking capabilities. Containers primarily rely on the use of namespace and cgroups to provide isolation, which enables users to run numerous applications in isolation on different containers on the same host. Additionally, due to the system level of abstraction, a container can freely move between host machines that support the container's runtime environment. Docker is the leading container solution widely accepted in industry. Docker has gained widespread acceptance in the recent years as can be seen by the support in resource managers and orchestration frameworks like Apache Mesos and Kubernetes. Another container technlogy that has gained the attention of the community is Singularity. It has been developed for scientific applications keeping the HPC eco-system as the focus. In the following subsections, we introduce these container mechanisms and their associated capabilities.

2.1 Docker Container

Docker can be considered a high-level user-space Linux utility, which can build, run and ship containers across hosts. Docker provides an isolated runtime environment. It is a lightweight Virtual Machine (VM) that has a networking configuration under a subnet and dedicated computational resources. A traditional VM abstracts the underlying hardware from guest OS, whereas Docker container provides one or more levels of abstraction by hiding the underlying host OS. Unlike other container solutions, Docker provides a virtual network, on top of the host machine, which can connect all containers to provide a convenient and secure inter-container communication.

2.2 Docker Swarm Mode

While containers provide a flexible packaging solution for building and shipping applications, additional tools are needed to manage the orchestration of multiple containers when running a distributed application. Docker Swarm addresses this need. It provides native support to manage the Docker container orchestration. This orchestration tool offers a software-defined overlay network across all participating host machines. In this setup, containers reside under the same virtual network and communicate with each other without any other networking configurations. An application can run as part of a swarm service, which can be scaled up and down as required. Swarm also can provide an attachable overlay network where containers from any host can be connected during creation, and all attached containers can be part of the same network.

2.3 Singularity Containers

Singularity has gained traction in the scientific community for its in-built support to integrate with the Message Passing Interface

(MPI). Singularity provides an easy packaging mechanism for applications along with a user friendly runtime execution environment. Singularity is designed to execute containers like a normal process in the host machine. This feature makes the integration of Singularity with the HPC schedulers easy. A fundamental difference with Docker container is the file format Singularity uses to store the image. Unlike Docker, which uses multiple layers, Singularity stores the entire image as one large file. Singularity hub provides the repository to save and maintain public Singularity container images.

2.4 InfiniBand

InfiniBand is well known for providing high throughput and low latency communication in distributed and parallel applications. For simplification, InfiniBand uses two channel adapters (CA) (1) Host CA (HCA) and (2) Target (TCA). Among the two channel adapters, only HCA provides visibility of the underlying hardware and software necessary for communication. The OpenFabrics Alliance controls the standard for developing software stack for the RDMA through InfiniBand. Vendors design and implement custom Verbs interfaces following the Verbs' specifications, which aims to address the interfaces that abstract the hardware components.

3 EXPERIMENTAL SETUP

We acquired six bare metal nodes from the Chameleon Cloud platform equipped with Mellanox InfiniBand interconnect. Each node consists of 48 cores and 128GB of RAM. Initially, each MPI benchmark was profiled to understand its characterization. For our experiments, a bare-metal execution is considered the baseline performance. We then use it to derive the respective performance variation for each containerization approach. Our set of benchmarks is composed of HPCG and miniFE for measuring the computational work, OSU-Micro benchmarks for measuring latency, and KMI Hash to profile memory bound applications. Each benchmark is profiled using experimental setups described in Table 3. We conducted ten iterations of each experiment and report the average.

Benchmarks	Description
HPCG	High Performance Conjugate Gradient
MiniFE	Unstructured finite element solver
OSU	Latency over MPI ranks
KMI Hash	Memory intensive integer operation

Table 1: *Software Stack and Version*

Software	Version
Linux	CentOS 7.4.1708
Open MPI	Open MPI-3.0.0
Infiniband	ConnectX3
Drivers	Mellanox OpenFabrics Enterprise Distribution

Table 2: *Software Stack and Hardware Components*

3.1 Bare Metal Nodes + InfiniBand (IB)

Cloud bare metal nodes with InfiniBand hardware were used int this setup. Open MPI 3.0.0 was configured with Mellanox interconnect driver, *ConnectX3*, for accessing InfiniBand hardware to facilitate RDMA communication. Each MPI benchmark in Table 1 was installed separately on each node. Experimental results for each benchmark obtained by this configuration is considered as the base, and performance deviation of each containerization approach.

3.2 Docker: one container per node, host network + InfiniBand

Figure 1 shows the required configuration for this setup. We configured one Docker container on each host node, where each container used the host network and shared the ip address of the host node. A new *userid*, "mpi" was added to all the container images along with an ssh daemon configured to run on port 9100, so that it is different from the default port 22 used by the host's sshd daemon. For mpirun, hostfile contained the ip addresses of the participating host nodes and the host network was used. However, the RDMA communication of MPI ranks was made possible by direct mapping of the InfiniBand devices of host node to the container. At the time of container deployment, each container was mapped to the host with the required Mellanox InfiniBand devices. All the drivers were made available via the container by commands shown in Listing 1. In order for the mapping of the devices to be accessible, containers needed to be started in privileged mode to access the devices of their host machine.

Listing 1: Launching containers on different host nodes with custom sshd port mapped to host node

```
$ docker run --privileged
-itd -p 9100:9100
--name host_container
--device=/dev/infiniband/rdma_cm
--device=/dev/infiniband/uverbs0
--device=/dev/infiniband/ucm0
--device=/dev/infiniband/umad0
--ulimit memlock=-1
--network=host
sciencecontainer/ompi3-infiniband:base
```

3.3 Docker: one container per node, overlay network + InfiniBand

Docker swarm mode enables users to abstract the underlying host machine topology through a virtual overlay network. This network can be spun across a few or a large number of host machines. Even though the physical location of each container may be different, as they all may not share the same host machine, they all share the software-defined network (SDN) and subnet address. This setup enables direct communication across containers. Unlike the previous approach in section 3.2, in this approach each container has a different IP address under the same subnet address. So, containers do not need to communicate over a non standard custom ssh port. It is important to note that we selected one of the containers as master-container and mapped its ssh port to the host node. The

Orchestration Method	*sshd Port*	*# of Containers per Host*	*IP Address*	*Network for mpirun and ssh*
Bare Metal (Section 3.1)	default (22)	0	host IP address	host network
Docker: Host Network (Section 3.2)	custom (9100)	1	host IP address	host network
Docker: Overley Network-I (Section 3.3)	default (22)	1	unique IP address	overlay network
Docker: Overlay Network-II (Section 3.4)	default (22)	n > 1	unique Ip address	overlay network
Singularity (Section 3.5)	default (22)	1	host IP address	host network

Table 3: Container Orchestration Methods

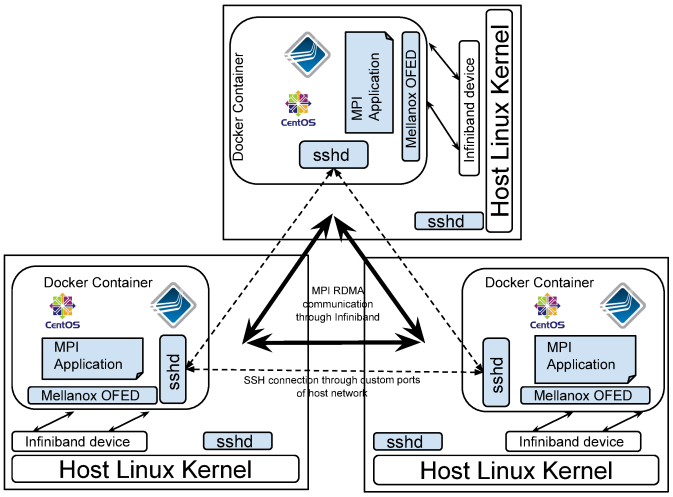


Figure 1: *One Docker container per host node is launched where each container can use the entire resource of the host node and host multiple MPI ranks. InfiniBand is used for MPI RDMA communication; however, TCP/IP over Ethernet is used for ssh and mpirun by running sshd on container's nonstandard port mapped to host node.*

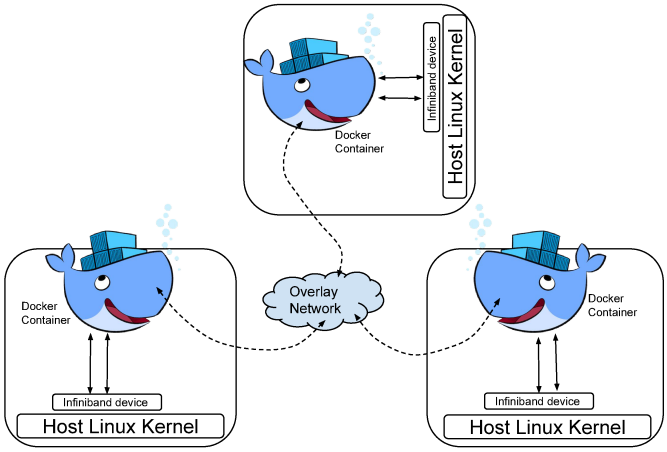


Figure 2: *One Docker container per host node is launched where containers from different hosts are connected to each other through Docker Swarm defined overlay network. ssh and mpirun use overlay network over standard ssh port, whereas for MPI RDMA communication, InfiniBand is used.*

master container is used as an entry point to start the MPI application from within the container. For mpirun, the *hostfile* contains the ip addresses of each container provided by the overlay network. RDMA communication across MPI rank was enabled as the host node's InfiniBand drivers were mapped as explained in the section 3.2.

Listing 2: Creating custom overlay attachable network

```
$ docker network create
--driver overlay --subnet 10.10.0.1/24
--attachable custom-network
```

3.4 Docker - Multiple containers per Host and 'n' MPI Ranks per Container

In this approach we launched multiple containers per host node and we split the MPI ranks among containers across nodes. As described in Section 3.3, each container is part of the Docker Swarm overlay network and for *mpirun* the *hostfile* contains the ip addresses of the containers. A configuration similar to the one mentioned in Section 3.3 was used while launching the containers via RDMA communication across MPI ranks and also for starting the MPI application.

Initially we launched one container for one MPI rank and then started growing the number of ranks per container while keeping the total number of ranks same. Containers were equally divided among host nodes in the cluster and each container hosted equal number of MPI ranks. The performance difference in this setup is dependent on the number of ranks per container.

Listing 3: creating Singularity images from Docker image

```
$ sudo singularity build
ompi3-infiniband-base.simg
docker://sciencecontainer/ompi3:base
```

Listing 4: Launching Singularity containers using host MPI libraries

```
$ mpirun -mca mtl_mxm_np 0
--mca btl openib -np 72
--map-by node
--hostfile ~/hostfile
singularity exec
ompi3-infiniband-base.simg
<path to executable inside container>
<parameters>
```

3.5 Singularity Container

We used MPI libraries of the host node to run applications through the Singularity containers. Running application through Singularity is like running the application on the host node. Singularity provides a way to create singularity images from existing Docker image layers. We used our Docker MPI images to create singularity images for the experiments.

4 EVALUATION

4.1 HPCG -High Performance Conjugate Gradients

We ran the HPCG benchmark on our experimental cluster with 72 MPI ranks for all the approaches mentioned in Table 3. In Figure 3a, our results show that the relative performance overhead of different containerization approaches is small when compared to traditional bare metal approaches. The approach (Section 3.2), *One Docker container per host* using host-network, yielded 0.65% overhead compared to bare metal. However, *one Docker container per host* with overlay-network (section 3.3) and Singularity showed performance reduction by 0.47% and 0.22% respectively.

In Figure 3b, we present HPCG performance when multiple MPI ranks are mapped with each Docker container. We ran *one container per rank* using 72 containers for a total 72 MPI ranks. We observed that the performance degraded by 50% compared to bare metal. However, when the number of ranks per container is increased from one to an amount equal or greater than 2, we observed that performance was similar to bare metal performance.

Figure 3c represents how the performance of the HPCG benchmark changes as we vary the number of nodes in a cluster. We varied the number of nodes from two (2) to six (6) and only executed 24 ranks throughout all the experiments. Our experimental results showed that, as expected, performance increased as the number of hosts in the cluster increased. From a two-node cluster to a four-node cluster, the performance increased by 37%. Additionally, after expanding the cluster by two more nodes, the performance increased by another 20%.

4.2 MiniFE - Finite Element mini-application

Figure 4a shows the performance of MiniFE benchmark with the various approaches when it ran with 96 MPI ranks. One container per host with the host-network yielded an overhead of 0.36% compared to bare metal. One container per host with overlay-network produced 0.15% degradation. On the other hand, when MiniFE ran with Singularity, we observed a 1.25% overhead in performance. Like HPCG, we observed similar behavior for MiniFE, when ranks are divided into multiple containers.

In Figure 4b, *one container per rank* approach produced 50% performance loss compared to bare metal, whereas when running two, four, eight, and sixteen ranks per container, the performance approached that of the bare metal setup.

We ran MiniFE with a fixed number of ranks (32 MPI ranks) while changing the size of the cluster from two nodes to 6 nodes. We observed that the performance expectedly improves as the number of nodes increases in the cluster. In Figure 4c, as the cluster size increased from two (2) nodes to four (4) nodes, the performance improved by 40%. Also, adding another two nodes into the cluster improved the performance by another 13% while keeping the number of ranks same.

4.3 OSU - Ohio State University Micro benchmarks

We chose *alltoallv* collective communication benchmark from the OSU benchmark suite to measure the latency across all the ranks when distributed across nodes. OSU can run 'n' processes where each sends 1/n of its allocated data to all the other ranks and receives a response back. We ran the latency benchmark for message size of 65536 Bytes. In Figure 5, our results show that the latency did not change significantly for any of our approaches. When we split the ranks into containers, *one container per rank* performance was within 0.82% of the bare metal performance, whereas four (4) ranks per container yielded an overhead of 0.72%. For the multi-node experiment, OSU ran with a fixed (32) number of ranks for an increasing number of hosts in which every variation outperformed the previous one – starting from two (2) hosts to six (6) hosts, with a final latency improvement of 63%.

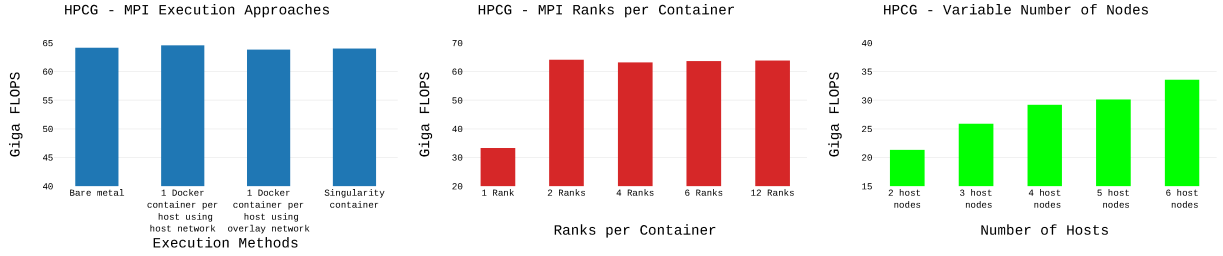
4.4 KMI Hash - M-mer Matching Interface benchmark

The purpose of KMI-Hash data-centric benchmark is to measure the performance of integer operation such as hashing. In Figure 6a, our experimental results show that Docker container approaches produce similar throughput compare to the setup with bare metal nodes. Singularity performs within 0.6% of the bare metal performance. In Figure 6b apart from *one container per rank*, other approaches did not show any significant overhead compared to the bare metal setup. In Figure 6c, multi-node execution for KMI hash yielded similar trends in results as the previous benchmarks. However, for KMI Hash the performance improved by 94% as the number of hosts increased from two nodes to six nodes while the number of ranks was fixed to 32.

4.5 Evaluating InfiniBand and Ethernet for different classes of MPI applications

We conducted experiments to study use of Containers with two different interconnects (1) InfiniBand for RDMA (2) Ethernet for TCP communication. In this approach, we considered three classes of MPI benchmarks (1)MiniFE as CPU bound (2)KMI Hash as memory bound and (3) OSU benchmark for latency. For the first two approaches (1) Ethernet on bare metal (Ethernet-BareMetal) and (2) Ethernet on Docker container on host network (Ethernet-Docker), both were used to measure the performance overhead compared to the third approach, which is bare metal with InfiniBand (InfiniBand-BareMetal). For the experimental evaluation, the average performance of 10 iterations was considered, and the results are presented in Table 4.

In Figure 7a, the benchmark MiniFE for Ethernet-BareMetal and Ethernet-Docker approaches performed within 1% and 1.4% of the InfiniBand-BareMetal approach respectively. This illustrates that CPU performance hit is minimal. However, Figure 7b shows a similar setup, but this time with a memory intensive benchmark, KMI

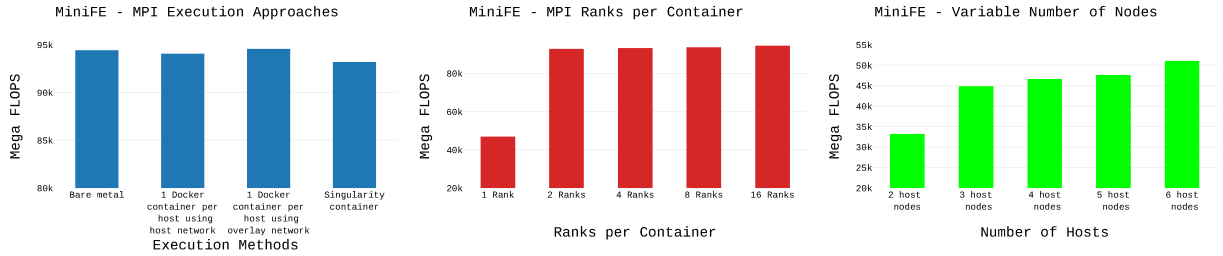


(a) *Performance comparison of HPCG benchmark with different approaches, running with 72 MPI ranks.*

(b) *HPCG benchmark performance comparison when the benchmark is running across multiple Docker container per host node. A total of 72 MPI ranks were used and distributed equally in varied number of containers. Each node hosted equal number of containers.*

(c) *HPCG benchmark evaluation when 24 MPI ranks were distributed across variable number of hosts in a cluster.*

Figure 3: HPCG Performance Evaluation with Different Execution Approaches.

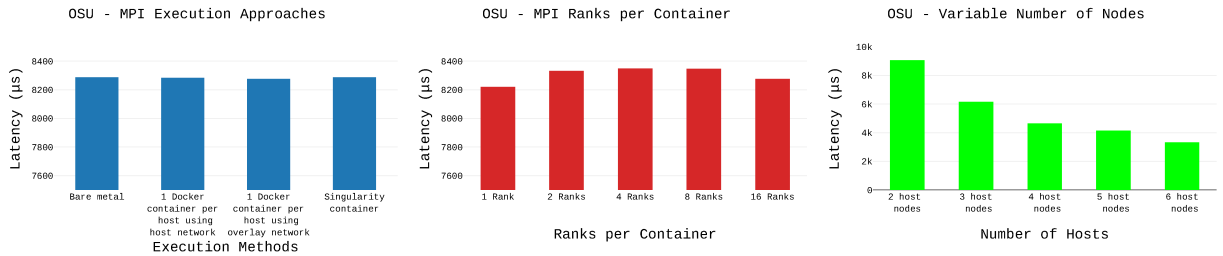


(a) *Performance comparison of MiniFE benchmark with different approaches, running with 96 MPI ranks.*

(b) *MiniFE benchmark performance comparison when the benchmark is running across multiple docker containers per host node. A total of 96 MPI ranks were used and distributed equally in varied number of containers. Each node hosted equal number of containers.*

(c) *MIniFE benchmark is evaluated when 32 MPI ranks were distributed across variable number of hosts in a cluster.*

Figure 4: MiniFE Performance Evaluation with Different Execution Approaches.

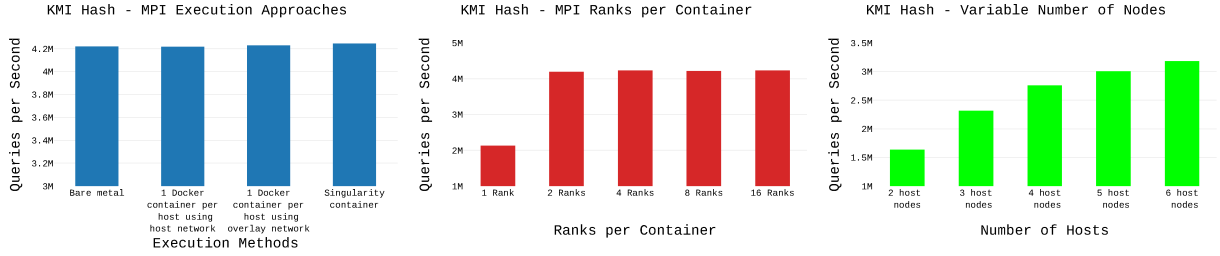


(a) *Performance of OSU latency benchmark benchmark with different approaches, running with 96 MPI ranks .*

(b) *OSU latency benchmark performance comparison when the benchmark is running across multiple Docker container per host node. In total 96 MPI ranks were used and distributed equally in varied number of containers. Each node hosted equal number of containers.*

(c) *OSU latency benchmark is evaluated with 32ranks, across variable number of hosts in the cluster.*

Figure 5: OSU (Latency) Performance Evaluation in Different Execution Approaches



(a) *KMI Hash benchmark benchmark with different approaches to compare the relative throughput for 72 MPI ranks.*

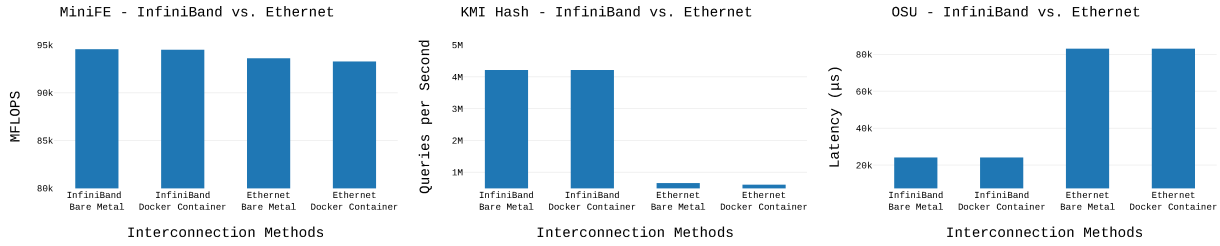
(b) *KMI Hash benchmark performance with different container to MPI rank ratio. In total, 72 MPI ranks were used and distributed equally in varied number of containers. Each node hosted equal number of containers.*

(c) *KMI Hash - Performance Evaluation with 24 MPI ranks on different number of hosts in the cluster.*

Figure 6: KMI Performance Evaluation with Different Execution Approaches.

Benchmarks	MPI Ranks	Interconnect Methods			
		InfiniBand-BareMetal	InfiniBand-Docker	Ethernet-Bare Metal	Ethernet-Docker
MiniFE (MFLOPS)	96	94557.7	94484.9	93612.5	93258.6
KMI Hash (queries/second)	72	4213375.4	4209725.1	653820.8	602075.2
OSU Latency (micro second)	96	24074.9	24064.7	82972.8	82994.4

Table 4: Performance of MPI benchmarks with different interconnect approaches



(a) *MiniFE - comparing performance over InfiniBand and Ethernet based interconnects for 96 MPI ranks*

(b) *KMI Hash - comparing performance over InfiniBand and Ethernet based interconnects for 72 MPI ranks*

(c) *OSU alltoallv - comparing performance over InfiniBand and Ethernet based interconnects for 96 MPI ranks*

Figure 7: Performance of MPI benchmarks with InfinBand and Ethernet interconnect for RDMA and TCP based MPI communication

hash. As expected, when using Ethernet-BareMetal and Ethernet-Docker approaches, we noticed an overhead of 84% and 85% respectively. This is a significant performance degradation compared to the InfiniBand-BareMetal approach. Unlike a CPU intensive benchmark, RDMA over InfiniBand interconnects outperforms the TCP based inter-process communication for a memory intensive benchmark such as KMI Hash. As expected, the OSU benchmark yielded better latency in the presence of InfiniBand. Figure 7c shows that Ethernet-BareMetal and Ethernet-Docker have close to 245% overhead compared to the InfiniBand-BareMetal approach. As presented in Figure 7, InfiniBand-Docker performed within 1% of the InfiniBand-BareMetal setup.

5 RELATED WORK

Docker is the widely used and supported containerization solution in the industry, but its adoption int HPC is hindered due to Docker's root escalation concerns. Azab et al. [1] developed a secure way of running Docker containers in HPC via a Slurm scheduler, without altering the underlying Docker engine and utilizing the full potential of Docker containers.

Apart from the security concerns due to root escalation, bandwidth and throughput for HPC jobs via Docker container has been another concern. HPC jobs require faster interconnection across ranks for better performance. Unlike Singularity containers, Docker

does not support InfiniBand (IB) interconnect as part of its architecture, but Chung et al. [2] deployed Docker on an IB setup and evaluated the performance of containers over IB with other visualization technologies. Chung et al.'s research also aimed to highlight the benefits of IB with Docker containers.

Younge et al. have defined a model for parallel MPI application DevOps for HPC systems to improve the development effort and reproducibility with the help of containers. They evaluated the feasibility of containers in HPC and showed the performance of Singularity containers on Cray systems [12].

In our previous work [7] [8], we have shown how Docker containers can be integrated with HPC environments and run MPI applications with cloud-enabled schedulers like Apache Mesos.

6 CONCLUSIONS

- Containers can be used to make HPC applications portable. They have proven to provide flexibility and maintainability for commercial applications executing on clouds.
- We conducted experiments to determine the performance of different benchmarks on *Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz* based cloud nodes. The performance of different containerization approaches are extremely close to bare metal. Different modes of running MPI application over a private cloud provides both flexibility and minimal performance overhead (less than 1%).
- Singularity provides direct support for MPI, and while Docker still does not provide full support for MPI, it is another choice developers and administrator can make. Docker provides more flexibility in terms of container placement with fine-grained resource allocation.
- Unlike Singularity, a Docker container needs to have InfiniBand interconnect drivers installed and mapped inside the container to enable fast communication.
- For MPI applications, splitting ranks per container with restricted resources to each container can be employed by Docker. This option is not available in Singularity containers.

A HEADINGS IN APPENDICES

A.1 Introduction

A.2 Background

A.2.1 Docker Container.

A.2.2 Docker Swarm Mode.

A.2.3 Singularity Containers.

A.2.4 InfiniBand.

A.3 Experimental Setup

A.3.1 Bare Metal Nodes + InfiniBand (IB).

A.3.2 Docker: one container per node, host network + InfiniBand.

A.3.3 Docker: one container per node, overlay network + InfiniBand.

A.3.4 Docker - Multiple containers per Host and 'n' MPI Ranks per Container.

A.3.5 Singularity Container.

A.4 Evaluation

A.4.1 HPCG -High Performance Conjugate Gradients.

A.4.2 MiniFE - Finite Element mini-application.

A.4.3 OSU - Ohio State University Micro benchmarks.

A.4.4 KMI Hash - M-mer Matching Interface benchmark.

A.4.5 Evaluating InfiniBand and Ethernet for different classes of MPI applications.

A.5 Related Work

A.6 Conclusions

A.7 References

ACKNOWLEDGMENTS

This work is partially supported by National Science Foundation, through the OAC-1740263 award.

REFERENCES

[1] Abdulrahman Azab. 2017. Enabling Docker Containers for High-Performance and Many-Task Computing. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 279–285. https://doi.org/10.1109/IC2E.2017.52

[2] Minh Thanh Chung, An Le, Nguyen Quang-Hung, Duc-Dung Nguyen, and Nam Thoai. 2016. Provision of Docker and InfiniBand in High Performance Computing. In *2016 International Conference on Advanced Computing and Applications (ACOMP)*. IEEE, 127–134. https://doi.org/10.1109/ACOMP.2016.027

[3] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. , 295–308 pages. http://dl.acm.org/citation.cfm?id=1972488

[4] Joe Mambretti, Jim Chen, and Fei Yeh. 2015. Next Generation Clouds, the Chameleon Cloud Testbed, and Software Defined Networking (SDN). In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*. IEEE, 73–79. https://doi.org/10.1109/ICCCRI.2015.10

[5] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[6] Nitin Naik. 2016. Building a virtual system of systems using docker swarm in multiple clouds. In *2016 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE, 1–3. https://doi.org/10.1109/SysEng.2016.7753148

[7] Pankaj Saha, Angel Beltre, and Madhusudhan Govindaraju. 2017. Scylla: A Mesos Framework for Container Based MPI Jobs. In *MTAGS17: 10th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*. Denver.

[8] Pankaj Saha, Madhusudhan Govindaraju, Suresh Marru, and Marlon Pierce. 2016. Integrating Apache Airavata with Docker, Marathon, and Mesos. *Concurrency and Computation: Practice and Experience* 28, 7 (5 2016), 1952–1959. https://doi.org/10.1002/cpe.3708

[9] Hideto Saito, Hui-Chuan Chloe Lee, and Ke-Jou Carol Hsu. 2016. *Kubernetes Cookbook*. Packt Publishing.

[10] Vinod Kumar Vavilapalli, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, Eric Baldeschwieler, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, and Hitesh Shah. 2013. Apache Hadoop YARN. In *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13*. ACM Press, New York, New York, USA, 1–16. https://doi.org/10.1145/2523616.2523633

[11] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. Springer, Berlin, Heidelberg, 44–60. https://doi.org/10.1007/10968987{_}3

[12] Andrew J. Younge, Kevin Pedretti, Ryan E. Grant, and Ron Brightwell. 2017. A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 74–81. https://doi.org/10.1109/CloudCom.2017.40