SAND2019-12753C

Enabling HPC workloads on Cloud Infrastructure using Kubernetes Container Orchestration Mechanisms

Angel Beltre[1], Pankaj Saha[1], Madhusudhan Govindaraju[1], Andrew J. Younge[2], and Ryan Eric Grant[2]

[1]SUNY Binghamton University, Binghamton, NY
Email: {abeltre1, psaha4, mgovinda}@binghamton.edu
[2]Sandia National Laboratories
Email: {ajyoung, regrant}@sandia.gov

***Abstract*—Containers offer a broad array of benefits, including a consistent lightweight runtime environment through OS-level virtualization, as well as low overhead to maintain and scale applications with high efficiency. Moreover, containers are known to package and deploy applications consistently across varying infrastructures. Container orchestrators manage a large number of containers for microservices based cloud applications. However, the use of such service orchestration frameworks towards HPC workloads remains relatively unexplored.**

In this paper we study the potential use of Kubernetes on HPC infrastructure for use by the scientific community. We directly compare both its features and performance against Docker Swarm and bare metal execution of HPC applications. Herein, we detail the configurations required for Kubernetes to operate with containerized MPI applications, specifically accounting for operations such as (1) underlying device access, (2) inter-container communication across different hosts, and (3) configuration limitations. This evaluation quantifies the performance difference between representative MPI workloads running both on bare metal and containerized orchestration frameworks with Kubernetes, operating over both Ethernet and InfiniBand interconnects. Our results show that Kubernetes and Docker Swarm can achieve near bare metal performance over RDMA communication when high performance transports are enabled. Our results also show that Kubernetes presents overheads for several HPC applications over TCP/IP protocol. However, Docker Swarm's throughput is near bare metal performance for the same applications.

***Index Terms*—Cloud Computing; Kubernetes; Container; HPC; Performance;**

I. Introduction

Throughout the computing industry, the advent of containers has fundamentally shifted how computational workloads are managed and orchestrated in distributed computing environments. Containers have emerged as a popular choice for application deployments in many cloud computing environments. This is due to their relatively simple resource management & isolation, high availability, portability, and improved efficiency over more conventional environments, including VMs. Containers are a natural fit with microservices based architecture in the cloud due to its seamless integration with cloud orchestrators and schedulers [1] [2] [3] for efficient resource managements. Microservices are loosely coupled, and independently maintained and deployed software modules that can also be auto-scaled as required. Given the paradigm shift towards microservices, the notion of container orchestration frameworks have quickly become of critical importance in today's distributed and cloud systems.

The deployment of container images is made possible via container orchestrators. Container orchestration tools have a mechanism to launch and manage containers as clusters or pods. The default orchestrator for Docker is Docker Swarm [4]. Other container orchestration tools include Google Container Engine [5], Amazon ECS [6], Mesosphere Marathon [7], Kubernetes, and Azure Container Service [8]. Among this list of container orchestrators, Docker Swarm and Kubernetes are the most widely known competing software platforms.

Concurrently, the use of High Performance Computing (HPC) has grown from humble beginnings in high-energy physics simulations to a wide variety of scientific endeavors including astrophysics, meteorology, chemistry, bioinformatics, and national security applications, to name a few. In this same regard, HPC's broader applicability towards advanced simulations and data analysis for enterprise and industrial applications has also grown. However, HPC applications have historically struggled with gaining adoption in the cloud, largely due to performance considerations at scale [9].

While Docker [4] is widely supported in cloud environments, it is not utilized in traditional HPC deployments due to concerns of root-level escalation in such a shared environment. However, a few other container technologies have been designed specifically for HPC. Modified container runtimes like Singularity [10], Shifter [11], and Charliecloud [12], all attempt to enable Docker containers to run on shared HPC resources. These mechanisms are meant to work with traditional HPC job submission tools. There is little work so far in studying container orchestration frameworks, specialized to manage microservices deployments, for executing HPC applications.

In this paper, we look to answer the question of whether HPC can be supported through the use of microservices container orchestration frameworks, rather than traditional batch queuing systems. Effectively, we look to provide HPC-as-a-Service using Kubernetes, a commodity service orchestra-

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

tion ecosystem. We compare the performance of container orchestration with Kubernetes, Docker Swarm, and traditional bare-metal, along with the options such as zero-copy RDMA-enabled communication, and traditional TCP/IP protocol. We use latency and throughput as comparison metrics and the Chameleon cloud [13] infrastructure as the testbed. Our goal is that these results, coupled with our detailed and optimized framework, can be used to inform the HPC community's application developers and users on how to best leverage and deploy their workloads using a combination of container technologies and microservice orchestration mechanisms.

The following are the key contributions of our work:

- We identify and present the configuration and settings required to deploy Kubernetes to support HPC applications with MPI based workloads. We explore the networking aspects and provide guidance on how to setup a Kubernetes cluster that can access the underlying hardware device specialized for HPC workloads.
- We evaluate the performance of Kubernetes and Docker Swarm when using the TCP/IP protocol, comparing to a bare metal deployment, using several characteristic HPC workloads.
- We present the design and evaluation for execution of MPI workloads on Kubernetes using the RDMA protocol and inter-pod communication.
- We evaluate and present the performance of bare metal versus containerized MPI applications orchestrated by Kubernetes and Docker Swarm. We use the results to identify the similarities and key differences between Kubernetes and Docker Swarm.

II. Background

We briefly summarize the key container and orchestration technologies for use with HPC applications. Our focus in this paper is on the Docker runtime engine. It is the leading container solution, and along with CRI-O, it is the default container solution for Kubernetes. Singularity, an HPC container solution, is the first Container Runtime Interface (CRI) that has been enabled in Kubernetes. In future work, we plan to extend our experimentation approach to Singularity, Shifter, and Charliecloud.

A. Containerization

1) ***Docker:*** Docker [4] containers are isolated applications, which are broken into smaller lightweight execution environments while sharing the operating system (OS) kernel. Docker creates a layer of abstraction to hide the underlying OS. Docker's networking capabilities support both *virtual networks* on top of the host network and *overlay networks*, which are commonly used to achieve a higher level of abstraction.

2) ***Singularity:*** Singularity [10] is a well known container solution within the HPC community. Unlike most containers, which focus on micro-service level virtualization (e.g., Rkt and Docker), Singularity's primary purpose is to provide application portability through operating system virtualization of namespaces, which is ideal for scientific computing ecosystems.

3) ***Shifter:*** Shifter [11] is a research and development (R&D) effort to bring containers into HPC ecosystems. Shifter is implemented to provide filesystem isolation through *chroot*, thus providing stricter security guarantees than the standard Docker image. It pulls and converts Docker images into a format known as *squashfs*, a compressed read-only file system format.

4) ***Charliecloud:*** Charliecloud [12] is a research effort by Los Alamos National Laboratory to execute Docker Containers in HPC system with minimal deployment requirements (i.e., non-root level access) by leveraging user namespaces within the Linux kernel.

B. Container Orchestrator

1) ***Docker Swarm:*** Docker Swarm can manage and orchestrate container deployment with in-built inter-container communication and software-defined networks. Swarm can optimize the underlying node's resource usage and distribute containers into different nodes to improve load balancing. Docker Swarm provides three basic strategies to distribute containers: *Spread*, *BinPack* and *Random*. Spread is the default strategy, which distributes the containers into different host nodes based on available resources. With BinPack, Swarm places containers one node at the time until it is fully occupied. The Random strategy places a container into a host node in a random order.

2) ***Kubernetes:*** Kubernetes is a container orchestrator that provides automation for running service containers. It provides a flexible way of scaling services running inside a container that require load balancing, fault tolerance, and horizontal scaling. Kubernetes provides direct support for Docker containers. Kubernetes uses *Pods* for better container management. A Pod is a logical envelope around a single container or multiple tightly coupled containers. The Kubernetes Pods are designed to be used with other container solutions such as Docker, Rkt [14], and runC [15]. There are two main requirements, (1) inter-container communication and (2) hardware device access, for supporting MPI based HPC applications on Kubernetes. Flannel [16] provides a mechanism to create a Software Defined Network (SDN) such that all pods created in any physical host can reach each other via unique IP addresses. Access to the host machines' hardware devices is essential for HPC specific network interconnects. Architectural components for Kubernetes can be categorized into two major categories: Master and Node Components. Kubernetes uses a *Master* node to serve as the primary global decision maker in the cluster. Four different components live in the master as presented in Figure 1: (1) Kube-apiserver, (2) Etcd, (3) Kube-scheduler, and (4) Kube-controller-manager. All the cluster data is stored and backed up by the distributed key-value store, *etcd*. Etcd can only be accessed through Kubernetes API server to prevent unsecured cluster access. Kube-scheduler assigns pods based on their resource requirements as well as general specifications.

Attributes	Kubernetes	Docker Swarm
Installation	OS specific setup, but cloud providers enable automatic configuration for enterprise customers.	Docker CLI commands are used for deployment
Device Access	System Pods must be provided as a resource with vendors specific software configurations.	Containers in a Swarm can only access devices running in privileged mode.
Overlay Network	Inter-Pod communication can only be enabled through the deployment of an overlay network (i.e., Flannel).	A default overlay network can be enabled for inter-container communication.
Shared Volume	Tightly coupled applications inside containers, can be placed inside the same pod to share volumes.	Storage can be mapped to containers with custom configurations.
Platform Dependency	Pods are created independent of the container solution.	It is limited only to Docker containers.

TABLE I: ***Kubernetes and Docker Swarm*** *Comparison of essential attributes offered by Docker Swarm and Kubernetes.*

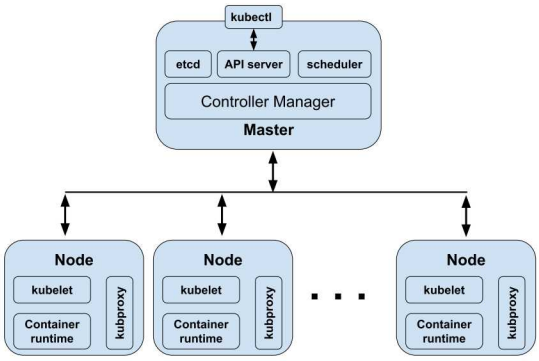


Fig. 1: **Kubernetes Components.** *The Kubernetes setup has at least three components: kublet daemon, Container runtime, and Kubeproxy for communication across nodes. The Master node contains the scheduling and administrative components.*

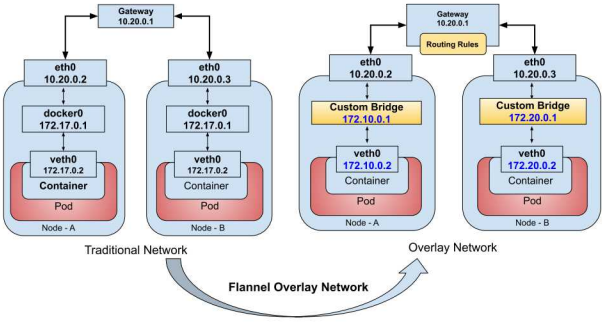


Fig. 2: ***Traditional Network to Overlay Network.*** *In conventional setups, each node in a cluster has its default docker bridge network. Docker containers (or Pods) are connected to the individual bridge of the host node and cannot communicate with containers (or Pods) residing in other host nodes. After applying an overlay network, across all the host nodes, containers and Pods that are created in any of the host nodes can communicate across the cluster.*

C. InfiniBand Interconnect

The use of InfiniBand (IB) [17] interconnect can provide high throughput and low latency across systems for distributed and parallel applications. IB is a standardized technology supported by major operating systems and vendors. IB comprises two channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). To enable communication through particular IB devices, the HCA makes different hardware visible at the user-level.

D. Network Instrumentation

Figure 2 shows a traditional network setup where each host has a private network, which is the same across all host machines. The IP addresses of each of the network interfaces are the same on each host except for the host network interface *eth0*. As illustrated in Figure 2 (Traditional Network), Docker creates the *docker0* network interface and every time a container gets launched on a virtual network, $veth\{0 \ldots N\}$ gets created. On the other hand, Kubernetes consolidates the network layout for a single host on a Kubernetes Pod over multiple containers by only creating a single virtual network that exchanges packets back and forth with the *docker0* interface. This approach provides a shared network interface with improved isolation, though at the cost of performance. In order to enable Pod-to-Pod communication, Kubernetes uses an overlay network as shown in Figure 2 (Overlay Network). Inter-pod communication, independent of Pods' physical location, is enabled using the Flannel [16] network package. The overlay network is deployed on Kubernetes cluster as a Pod network, which enables each pod to communicate with another through SDN with unique IP address over the overlay network.

E. Kubernetes and Docker Swarm Attributes

In Table I, we highlight the values of attributes that are required for both Kubernetes and Docker Swarm Cluster, to execute HPC workloads in a cloud infrastructure.

III. Kuberentes InfiniBand Setup

In Figure 3a, we present the configuration and setup of a device Pod for IB support. The architecture enables auto discovery of devices from Application Pods. The containers have full installations of Open-fabric software to enable zero-copy RDMA communication for the containers. For the pur-

Listing 1: A configuration *yaml* file for HCA device plugin

```
kind: ConfigMap
metadata:
  name: rdma-devices
  namespace: kube-system
data:
  config.json: |
    {
        "mode" : "hca"
    }
```

Listing 2: Integrating virtual network device as a resource.

```
resources:
  limits:
    rdma/hca: 1
    cpu: 48
```

poses of this paper, InfiniBand RDMA refers to InfiniBand send/recv data movement, not InfiniBand read/write methods as the codes used make use of MPI functions that utilize send/recv data movement. The steps to configure, install, and deploy a Device Pod container can be described as follows:

- **Device Configuration.** The virtual network device is configured using a *ConfigMap*, which is a Kubernetes object. The ConfigMap object is placed in the kube-system namespace along with its data as shown in Listing 1.
- **Device Deployment.** The device deployment is executed as a Kubernetes daemon object. It is also placed in the kube-system to enable access of resources to Kubernetes cluster nodes. The device plugin directories are mapped and the configuration map is set as shown in Listing 3.
- **Application Pod Configuration.** This particular Pod configuration consists of bridging the device to the application Pod. The application Pod demands at most one virtual networking device. In the example *yaml* file in Listing 2, we show the device being provided as a resource. Also, Listing 2 shows an example of how other resources (e.g. CPU, memory, GPU) can be specified.

Figure 3c shows the different networking levels that have to be set to ensure a full Kubernetes cluster setup. The Kubernetes Network is comprised of multiple layers of communication. For this setup, we list the sub-components of the networking layers that enable a Kubernetes cluster to work correctly.

Here, we describe the different Kubernetes networking communication layers:

- **Inter-container Networking.** Kubernetes' default container communication makes use of *localhost* network on default ssh-port network and the Pod's network *namespace*. In Figure 3c, in order to avoid running the MPI applications as a root user and on default ssh-port, we created a user inside the container for which a non-standard ssh-port was exposed to execute the application. Inter-container networking level essentially enables a container network interface for the communication.

Listing 3: *yaml* file to install and deploy a virtual network device.

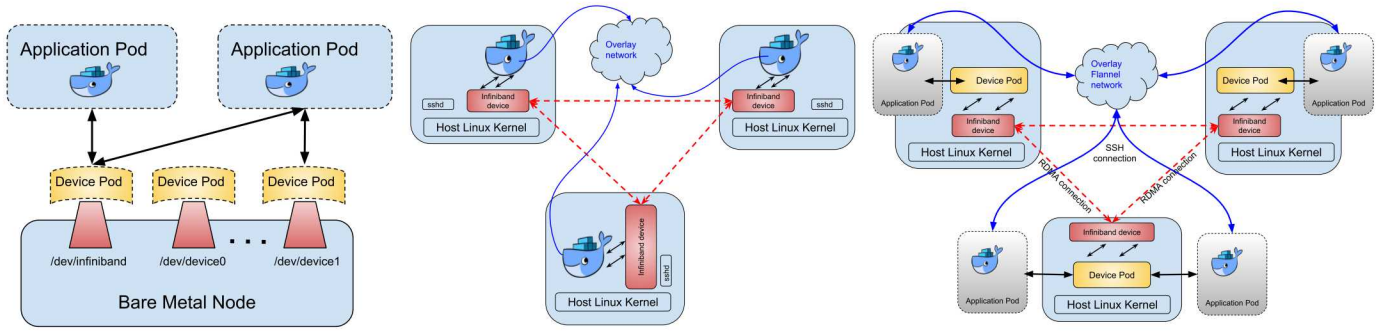
```
kind: DaemonSet
metadata:
  name: rdma-dev
  namespace: kube-system
...
- name: device-plugin
  hostPath:
    path: /var/lib/kubelet/device-plugins
- name: config
  configMap:
    name: rdma-devices
    items:
    - key: configuration.json
      path: configuration.json
```

- **Inter-Pod Networking.** Pods residing on the same host machines can communicate with each other as they are created under the same sub-network. However, Pods from different hosts are not reachable to each other as they are part of different networks of their respective host nodes. Inter-pod communication is enabled by using an overlay network. In Figure 3c, we use *Flannel* as our overlay network to enable inter-pod communication across host machines.
- **Application Pod.** In Figure 3c, the Application Pod shows how Docker containers reside on them. It shows how Device Pods, discussed in Figure 3a, connect with each Application Pod. It also shows the overlay network setup that ties Application Pods together across host machines.

Network setup is a crucial step to launch pods within a Kubernetes cluster. A Flannel network assigns a network to all pods on all nodes. Then, Docker bridge interface uses the Flannel provided network to create containers. Also, the latest MOFED user-space drivers were installed inside the Docker container to enable execution of MPI code from within the container.

In Figure 3b, we show the IB setup architecture of Docker Swarm. It presents the connections between nodes in terms of network (e.g. overlay network) and devices (e.g. InfiniBand). We list the necessary steps to get a Docker Swarm setup for MPI execution:

- **Overlay Network.** Overlay attachable network was created to maintain containers grouped together on a sub-network. So, containers are physically located in different host machines, but they share the same sub-net address space and the same SDN.
- **Container SSH Communication.** The communication between containers is enabled using a non-standard ssh-port and a unique IP address through the overlay network.
- **RDMA Enabled Containers.** The RDMA protocol was used to aid MPI execution of the containers residing across different host machines. InfiniBand devices are



(a) Device Pod (b) Docker Swarm Cluster (c) Kubernetes Cluster

Fig. 3: (a) *Kubernetes provides a unique way of accessing host nodes' hardware devices through vendor specific system pods.* (b) *An overlay attachable network to group container in a sub-network. A custom ssh port is exposed to enable inter-container communication across containers. All containers within the Swarm cluster have user-space IB drivers for MPI execution over RDMA protocol.* (c) *An overlay network was setup using Flannel to enable inter-Pod communication across nodes. A device Pod was created to allow device discovery from a container within a Pod.*

Listing 4: Accessing container within Pod to execute MPI on a Kubernetes cluster.

```
$ kubectl exec -it <pod> -- /bin/bash
$ mpirun --mca btl openib,self --mca btl_openib_if_include mlx4_0:1
  --hostfile hostfile --map-by node -np 128 ./osu_alltoallv
```

Listing 5: Accessing container and executing an MPI application on Docker Swarm cluster

```
$ ssh -i ssh/rsa mpi@localhost -p 9100
$ mpirun --mca pml ucx --mca btl openib,self --mca btl_openib_if_include mlx4_0:1
  --hostfile hostfile --map-by node -np 128 ./osu_alltoallv
```

mapped to the container at the time of deployment.

IV. Experimental Setup

For our experiments, we acquired six nodes from the Chameleon Cloud to set up a bare metal environment and also a container based environment. The purpose of these experiments is to investigate the efficacy of HPC with Kubernetes and to identify opportunities and areas for improvement. A full fledged scalability study is beyond the scope of this paper, due to the small cluster size. In future work, we plan to acquire larger clusters to perform scalability studies. In our experiments, we measured bare metal performance as the baseline for comparison with Kubernetes and Docker Swarm. We used the software and hardware components discussed in Table II. In addition, for all application execution, we selected the best performing network interface provided within each setup.

Software	**Version**
CentOS	7.6.1810 (Core)
Docker	18.09.1
Kubernetes	1.13.2
Open MPI	4.0.0
MLNX_OFED	4.5-1.0.1.0
InfiniBand Adapter	ConnectX-3

TABLE II: *Software, Hardware, and Network Stack*

In order to study container orchestrated MPI workloads on clouds such as Chameleon Cloud, we launched one container per machine for the Docker Swarm cluster and one Pod per machine for the Kubernetes cluster. We used a small cluster of six nodes to conduct our experiments, with each node consisting of 48 CPU cores, 128 GB of memory, 2TB of local storage, and IB hardware devices.

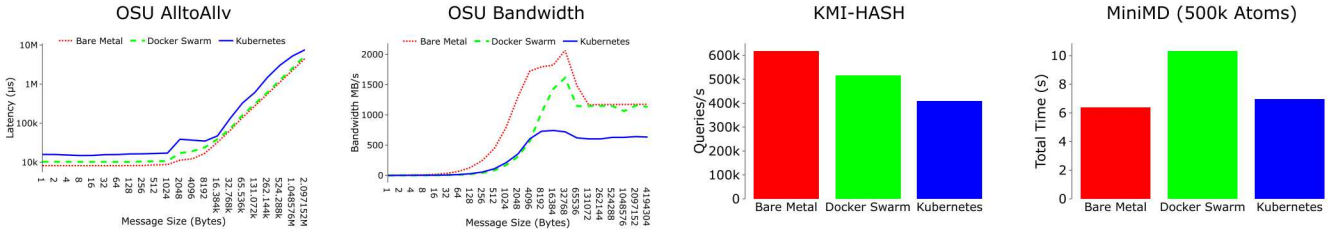
For our performance evaluation, we first executed all the benchmarks listed on Table III over an Ethernet interconnect on Chameleon cloud. Then, we executed the same set of benchmarks over RDMA protocol with the same settings. Both interconnects were orchestrated with three different environment setups: Bare Metal, Docker Swarm, and Kubernetes.

V. Performance Evaluation

1) ***OSU AlltoAll Latency****:* To study multi-node latency, we chose the AlltoAllv collective from the OSU benchmark suite of MPI benchmarks. OSU AlltoAllv communication is designed to have each MPI rank send a portion of its data to every other MPI rank, which is a global transposition operation acting on sub-portions of a particular data set. In essence, AlltoAll spreads 128 MPI processes across six host machines. Then, it performs a ping-pong between a sender and receiver. In Figure 4a, Kubernetes shows an overhead resulting in 4x performance loss in comparison to Docker Swarm over TCP/IP. We observe that as the message size increases from 8KB to around 1 MB for Docker Swarm, the

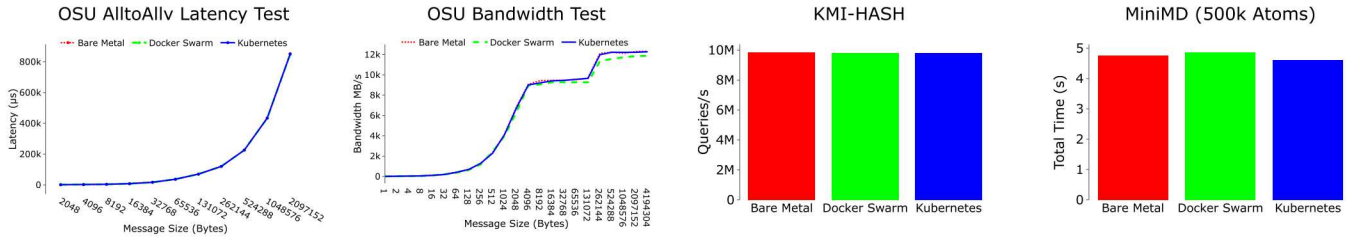
Benchmarks	Description	Metrics Measured
HPCG [18]	High Performance Conjugate Gradient	Throughput
MiniFE [19]	Unstructured finite element solver	Throughput and run time
OSU AlltoAllv [20]	Latency test of a set of ranks sending and receiving data	Latency
OSU Bi-directional [20]	Bandwidth test between two adjacent nodes	Bandwidth
KMI-Hash [21]	Evaluates hashing of architecture integer operations	Memory
HPL [22]	Computation of a dense n by n system of linear equations	Throughput
SNAP [23]	Discrete ordinates neutral particle transport	Run time and Memory
MiniMD [19]	Parallel molecular dynamics	Run time
MiniAMR [19]	Adaptive Mesh Refinement	Memory

TABLE III: *Benchmarks, descriptions, and metrics collected.*



(a) Latency (Log scale) (b) Bandwidth (c) Memory (d) Run time

Fig. 4: ***TCP over Ethernet (128 MPI processes, 6 hosts):*** *Latency (a) and bandwidth (b) evaluation results for respective OSU benchmarks. Queries/sec is evaluated for memory intensive KMI Hash (c) benchmark and runtime evaluation for MiniMD (d). Kubernetes presents a latency bottleneck of 4x in comparison to bare metal. For bandwidth, Kubernetes presents a visible overhead of 59.34% for a 8192-Bytes message in comparison to bare metal.*



(a) Latency (b) Bandwidth (c) Memory (d) Run time

Fig. 5: ***RDMA over InfiniBand (128 MPI processes, 6 hosts):*** *Latency (a) and bandwidth (b) evaluation results for respective OSU benchmarks. Queries/sec is evaluated for memory intensive KMI Hash (c) benchmark and runtime evaluation for MiniMD (d). Both Docker Swarm and Kubernetes for all the benchmarks present a performance deviation of 1% in comparison to bare metal.*

latency overhead, when compared to bare metal, improves from 66.77% (8 KB) to 15.95% (1 MB). However, in Figure 5a for InfiniBand, we ran the benchmark for message size of 2MB and it shows an average latency deviation of less than 1% for all the message sizes for both container-based setups, when compared to bare metal. This demonstrates the point at which the communication becomes wire data rate limited rather than host latency limited for Docker Swarm.

*2) **OSU Bidirectional Bandwidth**:* The OSU Bi-directional Bandwidth test over Ethernet sends back-to-back messages and waits for a response. It allows the measurement of the aggregated bandwidth between two nodes. Figure 4b measures and tests inter-node bandwidth using OSU Bidirectional Bandwidth over Ethernet connection on bare metal, Docker Swarm, and Kubernetes setups. Both Docker Swarm and Kubernetes experience similar bandwidth for message sizes up to 4KB. Docker Swarm achieves better performance as Kubernetes stagnates. For message sizes greater than 4KB, Kubernetes is outperformed by Docker Swarm, as Docker Swarm achieves a bandwidth closer to the Bare Metal. For example, for 4 KB message size the results show that both Docker Swarm and Kubernetes exhibit a slower performance in comparison to bare metal of 67.39% and 64.97% respectively. For a larger message of 8KB the overhead was about 42.14% and 59.34% for Docker Swarm and Kubernetes respectively. However, in Figure 5b, we can observe that bandwidth for both container-based setups have an average overhead of about 2% in comparison to bare metal for all messages.

*3) **KMI-Hash Evaluation**:* We used the KMI-Hash benchmark to characterize the behavior of memory intensive applications running on Docker Swarm and Kubernetes over Ethernet. We focused our attention on the number of queries that can be executed per second. In Figure 4c, both Docker Swarm and Kubernetes show a performance overhead of 16.58% and 33.96% respectively in comparison to bare metal. This performance variation is associated with the virtual network interfaces created on Docker Swarm and Kubernetes. In Figure 5c the execution over InfiniBand, however, yields an overhead of less than 1% for both container-based solutions in comparison to bare metal.

*4) **MiniMD Evaluation**:* MiniMD is a parallel molecular dynamics (MD) mini-application mainly used for testing purposes across different HPC systems. For MiniMD, the results in Figure 4d show the execution overhead using one thread per communication between communicating ranks. Docker Swarm and Kubernetes present overheads of 61.76% and 9.37% respectively in comparison to the bare metal execution time. However, unlike MiniMD over Ethernet, we notice that in Figure 5d the InfiniBand support amortizes the overhead presented by Docker Swarm and Kubernetes, as they have an average of just 1% overhead in comparison to bare metal.

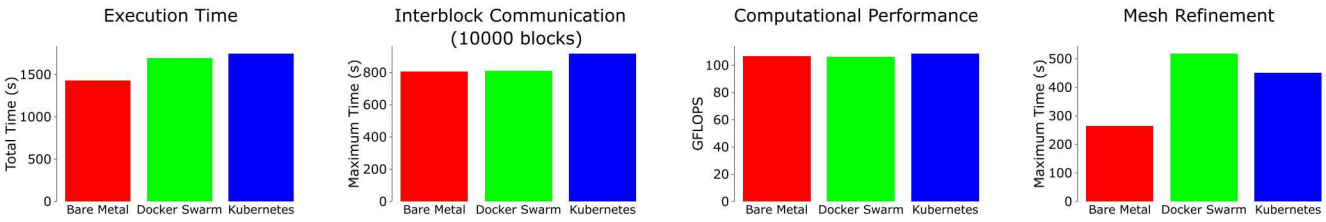
*5) **MiniAMR Evaluation**:* MiniAMR is a mini-application designed to carry out stencil calculations on a unit cube, which can be emulated in different bodies in space. We modified input parameters of two spheres to run on 128 cores for 100-time steps with x=4, y=4, and z=8 in each direction (i.e., 4x4x8=128). Over Ethernet, Figure 6a shows the results for MiniAMR over the different environment setups. We show the total execution time of the application where Docker Swarm and Kubernetes environment setups yielded an execution time overhead of 18.41% and 22.01% respectively in comparison to bare metal setup. Then, in Figure 6b, we present the interblock communication results (- $-max\_blocks = 10000$ per processor), which is an essential metric to understand the application's overall performance as about 47% to 59% of the overall execution time is spent executing communication tasks on all the environment setups. The results of the interblock communication for 10000 blocks do not show a large bottleneck. Kubernetes presented a 14.02% overhead in comparison to bare metal, whereas Docker Swarm had negligible overhead. In Figure 6c, we show MiniAMR's throughput with an overhead of less than 1% for both container-based solutions in comparison to bare metal. Figure 6d shows that Mesh Refinement's (which handles the Block computations) execution time on bare metal outperforms Docker Swarm and Kubernetes by 96.17% and 70.41% respectively. Figures 7a, 7b, 7c, and 7d over InfiniBand show an overhead of less than 1% for all the metrics collected.

*6) **MiniFE Evaluation**:* To study the throughput performance further, we evaluated MiniFE for two problems sizes – (1) nx=ny=nz=512 and (2) nx=ny=nz=1024. The results presented in Figure 8a for problem size (1) shows that the amount of data transported across processes in MiniFE can directly affect the application when using a low throughput transport protocol such as TCP/IP. Docker Swarm environment performs within 2% of Bare Metal. Kubernetes has an overhead of 54.29%. Docker Swarm also provides similar interfaces as Kubernetes, but we only included the data for the best performing interfaces within each setup. For problem size (2), we observe that for Docker Swarm and Kubernetes, the overhead was reduced to 1.24% and 15.70% respectively in comparison to Bare Metal. We also observe that doubling the problem size from 512 to 1024 yielded slightly better results for all environment setups, especially Kubernetes, which yielded a 94.50% better performance than problem size (1). In addition, in Figure 9b for MiniFE, the runtime results show the overhead of Kubernetes for problem size (1) and (2). It posts an overhead of 118.75% and 18.62% respectively in comparison to the same problem sizes on bare metal, whereas Docker Swarm shows a performance degradation of about 2% for both problem sizes. Our results for MiniFE over InfiniBand indicate that there is negligible performance overhead for the two cloud container orchestrators. Figures 9a and 9b present an overhead below 1% when compared to bare metal setup for both problem sizes.

*7) **HPCG Evaluation**:* We chose HPCG as it provides features that have a large impact on the application's overall performance. For HPCG, for a cube of 160 points, each individual MPI process (128 MPI processes) is in charge of performing computations for the 160 points. In addition, each environment setup consumes 374.91 GB of memory (i.e., 2.92 GB of memory per process). In Figure 8c, we can observe that over TCP/IP, Docker Swarm achieves close to Bare Metal performance whereas Kubernetes has an overehad of about 13.15%. These results highlight the stress HPCG places on network bandwidth and latency. In Figure 4, Kubernetes shows a significant overhead over both individual network tests – this trend can also be when executing HPCG as shown in Figure 8c. In 9c, HPCG over InfiniBand yields interesting results as Docker Swarm shows an overhead of 4.19%, which is more than the 2.2% deviation it has with TCP. Also, Kubernetes reduced its overhead by 2.84% (from 13.15% to 10.31%).

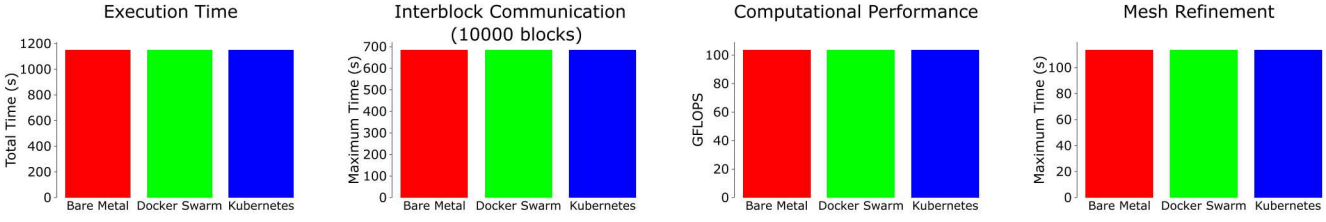
*8) **SNAP Evaluation**:* We executed SNAP with no OpenMP threads and also with 4 OpenMP threads per process with 128 MPI processes. In Figure 8d. with no OpenMP threads, SNAP shows an overhead in runtime for Docker Swarm and Kubernetes of 5.40% and 26.49% respectively. Kubernetes performs better when no OMP threads are used as there is a reduction in communication and computation. When multiple OMP threads are used for a single process on a single core, both Docker Swarm and Kubernetes present an increase in runtime of 8.68% and 46.64% respectively in comparison to bare metal (i.e., with 4 OMP threads per process or 512 threads total). In Figure 9d, SNAP achieves nearly bare metal performance for InfiniBand.

*9) **HPL Evaluation**:* Due to the compute intensive feature of HPL, with an Ethernet network connection we chose small P and Q values (i.e., HPL grid) to test performance. Note that the performance and scalability of HPL has limitations



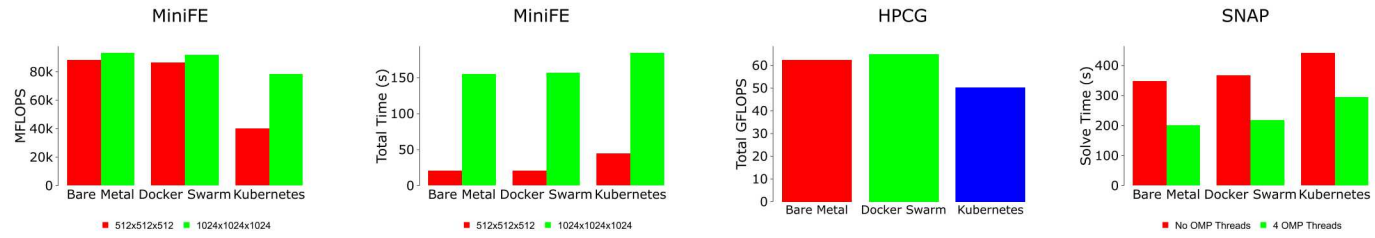
(a) Run time (b) Communication (c) Throughput (d) Run time

Fig. 6: ***MiniAMR Evaluation over Ethernet (128 MPI processes, 6 hosts):*** *Throughput and run time is evaluated for MiniAMR benchmark. Runtime is evaluated considering interblock communication and mesh refinement.For Mesh Refinement, Kubernetes has a performance overhead of 70.41%, whereas Docker Swarm presents an overehad of 96.17%.*



(a) Run time (b) Communication (c) Throughput (d) Run time

Fig. 7: ***MiniAMR Evaluation over InfiniBand (128 MPI processes, 6 hosts):*** *Throughput (c) and run time is evaluated for MiniAMR benchmark. Runtime is evaluated considering interblock communication and mesh refinement. Both Docker Swarm and Kubernetes for all the benchmarks present a performance deviation of 1% in comparison to bare metal.*



(a) Throughput (b) Run time (c) Throughput (d) Run time

Fig. 8: ***TCP over Ethernet (128 MPI processes, 6 hosts):*** *Throughput (a) and run time (b) evaluation for MiniFE benchmarks. Throughput is evaluated for HPCG (c) benchmark and run time evaluation for SNAP (d) for 1 and 4 threads per process. For a problem size of 512x512x512, the Docker Swarm environment performs within 2% of Bare Metal, whereas Kubernetes environment has an overhead of 54.29%.*

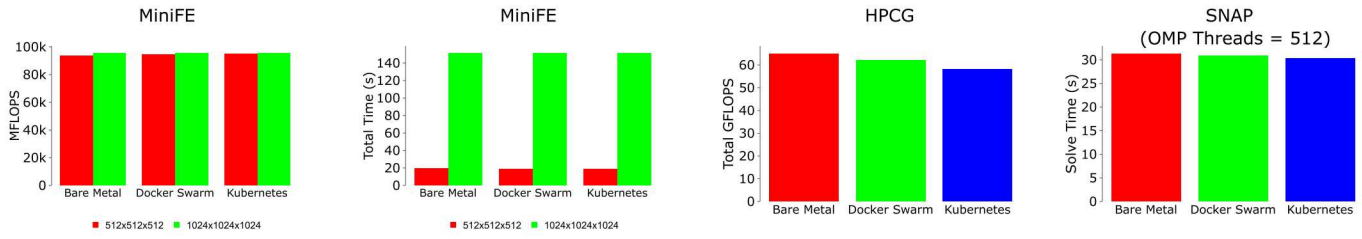
since the chosen grids are small. Another attribute that is important to run HPL is problem size (i.e., N) – the size of any problem should be the largest problem size that fits in memory, without creating memory swapping limitations. Lastly, another important variable for HPL is block size (i.e., NB), a flag used for data distribution. In our HPL execution, we used a problem size of 84000 and block size of 96, with memory consumption of about 80% of the total across the 6-node cluster. For our P and Q, we chose a grid of 4 by 32 respectively. In Figure 10a, we present the performance of HPL over InfiniBand with both container-based setups performing within 1% of bare metal.

VI. Related Work

The cloud and HPC convergence has gained a lot of traction as a topic of research. Cloud technologies has gained enough traction in recent time into different communities such a way that it enables the adoption of these cloud based solutions into their respective domains [24] [25]. The literature survey in this paper is limited to related articles, as research on use of Kubernetes in HPC is in its early stages.

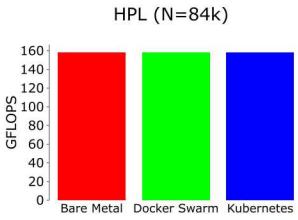
Younge et al. [26] evaluated the feasibility of different container mechanisms to improve the development effort and DevOps for MPI applications in HPC systems. According to their results, virtual machines, and containerized applications on cloud nodes (no bare metal nodes were used) present a substantial performance overhead compared to high end HPC clusters. However, the the performance of MPI applications on a Cray supercomputer using Singularity containers demonstrated near bare metal performance.

Saha et al. [27] evaluated the performance of HPC ap-



(a) Throughput (b) Run time (c) Throughput (d) Run time

Fig. 9: ***RDMA over InfiniBand (128 MPI processes, 6 hosts):*** *Throughput (a) and run time (b) evaluation for MiniFE benchmarks. Throughput is evaluated for HPCG (c) benchmark and run time evaluation for SNAP (d) for 4 threads per process. Both Docker Swarm and Kubernetes for MiniFE and SNAP present a performance deviation of 1% in comparison to bare metal.*



(a) Throughput

Fig. 10: ***HPL Evaluation over Ethernet (128 MPI processes, 6 hosts):*** *Throughput is evaluated for a small problem size. Both Docker Swarm and Kubernetes present a performance deviation of 1% in comparison to bare metal.*

plications in a cloud setup wherein containerized applications were run with MPI ranks distributed across multiple containers using Docker Swarm. The results showed that the performance deviation of Docker Swarm was negligible compared to bare metal performance. In this paper, we explained the components of Kubernetes and compared its HPC workload performance with Docker Swarm and bare metal setups.

Saha et al. [28] showed how MPI applications can be orchestrated in an Apache Mesos cluster with just the use of Docker Swarm as the container orchestrator. They provided insights regarding how cloud based resource managers like Apache Mesos [1] can be used for HPC workloads using a policy based approach [28] for scheduling of MPI ranks based on the nature of tasks (Network or CPU intensive). InfiniBand performance versus TCP in the context of bare metal solutions has been evaluated for HPC. Grant et al. [29] evaluated different RDMA options versus TCP and demonstrated its impact on commercial data center applications [30]. Balaji et al. have evaluated traditional TCP versus IB alternative communication sockets-compatible libraries for 10G Ethernet [31]. Rashti et al. performed a an evaluation of TCP versus IB and Myrinet networks [32]. In this paper, we focus on the architecture, configuration, and evaluation of Kubernetes for use with scientific workloads in clouds such as Chameleon.

Another related work is an effort to enable HPC workloads execution on Kubernetes is *kube-batch* [33]. It is designed to support scheduling and OpenMPI execution. In our work, we focused on understanding the performance of Kubernetes for HPC workloads using its default scheduler.

VII. Conclusions

In this paper, we used a diverse set of MPI applications to evaluate their performance and determine the feasibility of executing them in two different cloud configurations: bare metal and container-based. Our container-based setups consisted of Docker Swarm and Kubernetes. Our metrics included memory usage, bandwidth, and latency.

For TCP/IP, both container solutions show substantial overhead for memory, network bandwidth, and latency. For latency, in Figure 4a, OSU AlltoAll latency test execution on Kubernetes shows an overhead of 4x compared to Docker Swarm. For bandwidth, in Figure 4b, OSU Bi-directional bandwidth test execution on Kubernetes and Docker Swarm for a message size of 8192 Bytes, show an overhead of 42.14% and 59.34% respectively. In Figure 8, we observe that a throughput execution of HPCG over TCP/IP with Kubernetes results in an overhead of 13.15% in comparison to bare metal.

For InfiniBand, we learned that for applications with low complexity such as OSU latency, OSU bi-directional bandwidth, SNAP, and MiniAMR the performance is within 1% of bare metal. For applications such as HPL, which is compute intensive and also low complexity, we observe for both Docker Swarm and Kubernetes a similar performance overhead of less than 1%. However, in Figure 9c for HPCG, InfiniBand transport reduces the overhead presented by Kubernetes in comparison to bare metal from 13.15% (55.29 GFlops) to 10.31% (58.26 GFlops).

We observed that there are opportunities when enabling a high performance transport in all the setups. We attribute the Kubernetes overhead to the fact that its whole network stack is virtualized. In a Docker Swarm setup, users have the flexibility to choose between virtualized and non-virtualized interfaces. We identified that it is crucial for HPC developers and architects to use an evaluation framework, such as the one we have developed, to study and make informed decisions on the configurations and setups to use for executing their workloads.

ACKNOWLEDGMENT

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. DOE National Nuclear Security Administration under contract DE-NA-0003525. This research was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

In addition, this work was supported in part by NSF grant OAC-1740263.

REFERENCES

[1] P. Saha, A. Beltre, and M. Govindaraju, "Exploring the fairness and resource distribution in an apache mesos environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 434–441. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00061

[2] A. Beltre, P. Saha, and M. Govindaraju, "Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters," in *IEEE Cloud Summit: 3rd IEEE International Conference on Cloud and Fog Computing Technologies and Applications*, August 2019.

[3] P. Saha, A. Beltre, and M. Govindaraju, "Tromino: Demand and drf aware multi-tenant queue manager for apache mesos cluster," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2018, pp. 63–72.

[4] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[5] [Online]. Available: https://cloud.google.com/kubernetes-engine/

[6] "Amazon ecs - run containerized applications in production." [Online]. Available: https://aws.amazon.com/ecs/

[7] "Marathon: A container orchestration platform for Mesos and DC/OS." [Online]. Available: https://mesosphere.github.io/marathon/

[8] Seanmck, "Choose the cloud platform designed for your container needs." [Online]. Available: https://azure.microsoft.com/en-us/overview/containers/

[9] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu, "Magellan: Experiences from a science cloud," in *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ser. ScienceCloud '11. ACM, 2011, pp. 49–58. [Online]. Available: http://doi.acm.org/10.1145/1996109.1996119

[10] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[11] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, 2015.

[12] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 36.

[13] J. Mambretti, J. Chen, and F. Yeh, "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)," in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*. IEEE, 2015, pp. 73–79.

[14] "Coreos." [Online]. Available: https://coreos.com/rkt/

[15] Opencontainers, "opencontainers/runc," Oct 2018. [Online]. Available: https://github.com/opencontainers/runc

[16] "flannel." [Online]. Available: https://coreos.com/flannel/docs/latest/

[17] T. Shanley, *Infiniband*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[18] J. Dongarra and P. Luszczek, "Hpcg technical specification," *Sandia National Laboratories, Sandia Report SAND2013-8752*, 2013.

[19] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[20] [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[21] M. Leininger, "Benchmark codes." [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[22] J. Dongarra, "The linpack benchmark: An explanation," in *Proceedings of the 1st International Conference on Supercomputing*. London, UK, UK: Springer-Verlag, 1988, pp. 456–474. [Online]. Available: http://dl.acm.org/citation.cfm?id=647970.742568

[23] R. J. Zerr and R. S. Baker, "Snap: Sn (discrete ordinates) application proxy," *Online: https://github. com/losalamos/SNAP. Accessed: Sep*, 2014.

[24] P. Saha, M. Govindaraju, S. Marru, and M. Pierce, "Integrating apache airavata with docker, marathon, and mesos," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 7, pp. 1952–1959, 2016.

[25] ——, "Multicloud resource management using apache mesos with apache airavata," *arXiv preprint arXiv:1906.07312*, 2019.

[26] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "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)*, Dec 2017, pp. 74–81.

[27] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *Proceedings of the Practice and Experience on Advanced Research Computing*. ACM, 2018, p. 11.

[28] P. Saha, A. Beltre, and M. Govindaraju, "Scylla: A Mesos Framework for Container Based MPI Jobs," in *MTAGS17: 10th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*, Denver, 2017.

[29] R. E. Grant, M. J. Rashti, and A. Afsahi, "An analysis of qos provisioning for sockets direct protocol vs. ipoib over modern infiniband networks," in *2008 International Conference on Parallel Processing-Workshops*. IEEE, 2008, pp. 79–86.

[30] R. E. Grant, P. Balaji, and A. Afsahi, "A study of hardware assisted ip over infiniband and its impact on enterprise data center performance," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 144–153.

[31] P. Balaji, H. V. Shah, and D. K. Panda, "Sockets vs rdma interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck," in *In RAIT workshop*, vol. 4, 2004, p. 2004.

[32] M. J. Rashti and A. Afsahi, "10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

[33] kubernetes sigs, "kube-batch," https://github.com/kubernetes-sigs/kube-batch, 2019.