High Performance MPI Library for Container-based HPC Cloud on InfiniBand Clusters

Jie Zhang, Xiaoyi Lu and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH USA 43210
Email: {zhanjie, luxi, panda}@cse.ohio-state.edu

***Abstract*—Virtualization technology has grown rapidly over the past few decades. As a lightweight solution, container-based virtualization provides a promising approach to efficiently build HPC clouds. However, our study shows clear performance bottleneck when running MPI jobs on multi-container environments. This motivates us to first analyze the performance bottleneck for MPI jobs running in different container deployment scenarios. To eliminate performance bottleneck, we propose a high performance locality-aware MPI library, which is able to dynamically detect co-resident containers at runtime. Through this design, the MPI processes in co-resident containers can communicate to each other by shared memory and Cross Memory Attach (CMA) channels instead of the network channel. A comprehensive performance study indicates that compared with the default case, our proposed design can significantly improve the communication performance by up to 9X and 86% in terms of MPI point-to-point and collective operations, respectively. The results for applications demonstrate that the locality-aware design can reduce up to 16% of execution time. The evaluation results also show that by the help of locality-aware design, we can achieve near-native performance in container-based HPC cloud with minor overhead. The proposed locality-aware MPI design reveals significant potential to be utilized to efficiently build large scale container-based HPC clouds.**

***Keywords*-Virtualization, Container, Cloud Computing, MPI**

I. Introduction

To meet the increasing demand for computational power, HPC clusters have grown tremendously in size and complexity. As the prevalence of high-speed interconnects, multi/many- core processors, and accelerators continue to increase, efficient sharing of such resources is becoming more important to achieve faster turnaround time and reduce the cost per user. Furthermore, a large number of users, including many enterprise users, experience large variability in workloads depending on business needs, which makes predicting the required resources for future workloads a difficult task. For such users, cloud computing can be an attractive solution that offers on-demand resource acquisition, high configurability, and high performance at a low cost. This demand has been evidenced by the plethora of vendors offering such solutions including Amazon, Google, Microsoft, etc.

Virtualization technology, as one of the foundations of cloud computing, has been developing rapidly over the past few decades. Several different virtualization solutions, such as Xen [1], VMware ESX/ESXi [2], and KVM [3], are proposed and improved by community. These hypervisor-based virtualization solutions bring several benefits, including hardware independence, high availability, isolation, and security. They have been widely adopted in industry computing environments. For instance, Amazon's Elastic Computing Cloud (EC2) [4], Google's Compute Engine [5] and VMWare's vCloud Air [6] utilize Xen, KVM and ESX/ESXi on their cloud computing platforms, respectively. However, their adoptions are still constrained in the HPC contexts due to their inherent performance overhead, especially in terms of I/O [7–9].

On the other hand, as a lightweight virtualization solution [10], container-based virtualization (such as Linux-VServer [11], Linux Containers (LXC) [12] or Docker [13]) has attracted considerable attention recently. It utilizes the system libraries and binaries on the host to achieve the isolation between containers. This use of the kernel across containers is similar to a hypervisor but does not allow running different guest operating systems. It thus makes the container-based solution a more efficient competitor to the traditional hypervisor-based solution. It is anticipated that the container-based solution will continue to grow and influence the direction of cloud computing.

A. Motivation

In the HPC context, there are several studies focusing on the performance characterization of container-based virtualization solution [14–16]. The results indicate that the container-based solution can deliver near-native performance for HPC applications. Based on these studies, we run the MPI-simple version of Graph 500 [17] with 16 processes and a problem scale of 20 and edge factor of 16 using default MPI library on a single host equipped with InifiniBand. We measure the time spent on Breadth-First Search (BFS) on a

*This research is supported in part by National Science Foundation grants #CNS-1419123, #IIS-1447804, #ACI-1450440 and #CNS-1513120.

2332-5690/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPP.2016.38

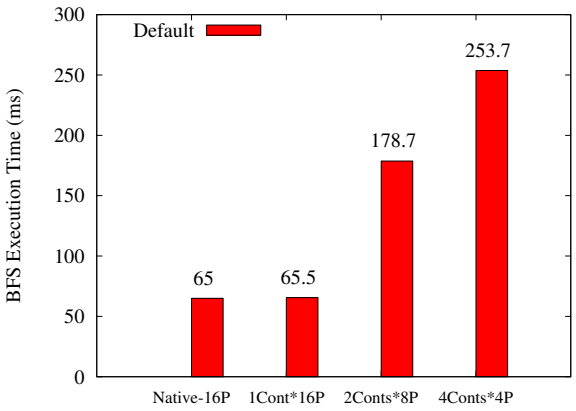


Figure 1: Execution Time of Graph 500 with 16 Processes using Default MPI Library under Different Container Deployment Scenarios

constructed graph [18]. As shown in Figure 1, the BFS time on one container is similar to that on the native environment. This is consistent with the conclusions of the above studies. Considering the characteristics of cloud computing environment, we increase the number of containers further and keep the total number of processes the same. We deploy two and four containers on that host, respectively. However, we find that the BFS time significantly increases as the number of container increases. These results motivate us to answer a challenging question: **Can we significantly improve the application performance when running across multiple containers per host so that it can deliver near-native performance for the different container deployment scenarios?** This further leads to the following broad challenges:

1) What are the fundamental performance bottleneck when running MPI applications on multiple containers per host on HPC cloud? Does the bottleneck lie in container runtime or MPI runtime?
2) Can we propose a new design to overcome the bottleneck and significantly improve the application performance on such container-based HPC cloud?
3) If so, how much performance benefits can be achieved? Can container delivers near-native performance for different container deployment scenarios?

B. Contribution

In order to improve the application performance under different container deployment scenarios, we need to understand their performance characteristics accordingly. In this paper, we first perform a detailed performance analysis using Graph 500. Through the analysis, we identify that the major performance bottleneck lies in the MPI communication performance across co-located containers. Further, we propose a design of high performance locality aware MPI library for container-based HPC cloud, which is able to dynamically detect co-resident containers before communication happens. In this way, shared memory and Cross Memory Attach (CMA) [19] based MPI communication can be executed across co-resident containers. Consequently, the communication performance can be significantly improved. Finally, we conduct comprehensive performance evaluations for our proposed design on Chameleon Cloud [20]. The results show that our proposed design can improve the performance of point-to-point and collective operations, and application by up to 9X, 86% and 16%, respectively. Compared to the native performance, the proposed design has a small overhead. The results also indicate that through our proposed design, container can deliver near-native performance (less than 5% overhead) for MPI applications running with different container deployment scenarios.

The rest of the paper is organized as follows. Section II mainly introduces two types of virtualization solutions, container privilege and cross memory attach technology. Section III analyzes and identifies the performance bottleneck based on the experiments we run in Section I-A. To eliminate the bottleneck, we propose and optimize our design of a high performance locality aware MPI library for container-based HPC cloud in Section IV. We conduct a comprehensive performance evaluation in Section V. We discuss the related work in Section VI, and conclude this paper in Section VII with future works.

II. Background

A. Container-Based Virtualization

Virtualization utilizes an intermediate software layer on top of an underlying system in order to provide abstractions of multiple virtual resources. In general, the virtualized resources are called virtual machines (VM) and can be seen as isolated execution environments. There are a variety of virtualization techniques. One of the most popular is the hypervisor-based virtualization, which includes Xen, VMware, KVM as its main representatives. As shown in Figure 2(a), the hypervisor-based virtualization, in its most common form, consists of a hypervisor, also called the virtual machine monitor (VMM), on top of a host operating system that provides a full abstraction of VM. In this case, each VM has its own operating system that executes completely isolated from the others. This allows, for instance, the execution of multiple different operating systems on a single host.

A lightweight alternative to the hypervisor-based virtualization is the container-based virtualization. In container-based virtualization, the host kernel allows the execution of several isolated user space instances that share the same kernel but possibly run a different software stack (system libraries, services, applications), as shown in Figure 2(b). Container-based virtualization does not introduce a layer of virtual hardware, however, it provides self-contained execution environments, effectively isolating applications that rely on the same kernel in the Linux operating system. Containers

are built around two core mature Linux technologies. First, cgroups (control groups) can be used to group processes and limit their resources usage. Second, namespace isolation can be used to isolate a group of processes at various levels: networking, filesystem, users, process identifiers, etc. Several container-based solutions have been developed, such as LXC, Googles lmctfy and Docker. Docker [13] is a popular open-source platform for building and running containers and offers several important features, including portable deployment across machines, versioning, reuse of container image and a searchable public registry for images. In addition, Docker gives users the flexibility to share certain namespaces with either the host or other containers. For example, sharing the host's process (PID) namespace allows the processes within the containers to see all of the processes on the system. And sharing IPC namespace can accelerate inter-process communication with shared memory segments, semaphores, and message queues. Nevertheless, the MPI library still needs modification to take advantage of these flexibilities. We deploy Docker to run containers in this paper.

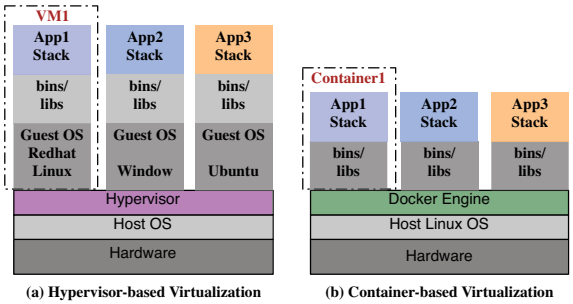


Figure 2: Hypervisor- and Container-based Virtualization

B. PCI Passthrough and Runtime Privilege

PCI passthrough allows giving access and control of the physical devices to guests: that is, you can use PCI passthrough to assign a PCI device (NIC, disk controller, HBA, USB controller, sound card, etc.) to a guest domain, giving it full and direct access to the PCI device. Similarly in container context, runtime privilege gives container access to all devices. For example, when the operator executes `docker run --privileged`, Docker will enable access to all devices on the host as well as set some configuration in SELinux to allow the container to have nearly all the same access to the host as processes running outside containers on the host. In this paper, we use privileged option to give container access to the InfiniBand device on the host.

C. Cross Memory Attach (CMA)

Cross Memory Attach (CMA) allows a destination process, given an address and size from a source process, to copy memory directly from the source process into its own address space via a system call. There is also a symmetrical ability to copy from the current process's address space into a destination process's address space. CMA allows intra-node communication in MPI applications to achieve better performance by a single copy of the message rather than double copies of the message via shared memory.

D. InfiniBand

InfiniBand [21] is an industry standard switched fabric designed for interconnecting nodes in HPC clusters. The TOP500 rankings released in November 2015 indicate that more than 47% of the computing systems (237) use InfiniBand as their primary high performance interconnect. Remote Direct Memory Access (RDMA) is one of the main features of InfiniBand, which allows software to remotely access memory contents of another remote process without any involvement at the remote side. Recently, InifiniBand has become popular on building HPC clouds, such as Chameleon Cloud. As container-based virtualization provides a lightweight alternative, building HPC cloud with InfiniBand and container seems a promising direction.

III. Bottleneck Analysis

In this section, we conduct a detailed performance analysis based on the experiments we run on Section I-A. From Figure 1, we observe three major characteristics:
(1) The BFS execution time is almost the same as that on the native host using only one container with the same number of process.
(2) However, the BFS execution time is significantly increased if we deploy two containers on that host, each with eight processes.
(3) Keeping increasing the number of container on the single host leads to further performance degradation.

In order to analyze the application performance bottleneck, we use mpiP [22] to profile the BFS execution time spent by the different container deployment scenarios on the following two aspects [18]:
(1) Communication: The BFS operation involves three communication patterns, which are asynchronous point-to-point communication (`MPI_Irecv`, `MPI_Isend`), polling of pending asynchronous messages (`MPI_Test`), and collective communication `MPI_Allreduce`.
(2) Computation: It is quantified as the time spent outside of any MPI calls.

Figure 3(a) shows the time-wise breakdown for the different container deployment scenarios. The results indicate that the BFS operation is highly communication-bound. On the native environment, 77% of BFS execution time is spent on communication. The proportion on one container case is similar with that on the native host. Once deploying two containers on the host, the communication proportion dramatically increases to 91%. The communication proportion increases further to 93% on four containers case. In addition, the computation time almost remains the same

on the different container deployment scenarios, which is around $17ms$. That is, the delay of communication time directly leads to the increase of BFS execution time.

Table I: Statistics on Message Transfer Operation

	Native	1-Container	2-Containers	4-Containers
CMA	960,450	960,450	640,422	286,787
SHM	168,302	168,302	112,259	50,624
HCA	0	0	376,071	791,341

It makes us to further explore which part of communication gets delayed for the case of two and four containers per node. The default MVAPICH2 implementation (similar for other MPI distributions) has three communication channels, which are shared memory (**SHM**) channel, **CMA** channel and network (InfiniBand **HCA**) channel. Figure 4 describes these communication channels for different container deployment scenarios. The communication goes through SHM channel if the communicating processes are co-located in the same container. To further optimize the communication performance, CMA channel is used when the message size exceeds the predefined threshold. If the communicating processes are across containers, then the HCA channel is selected by default. Figures 3(b) and 3(c) present the performance comparison among the above three channels with respect to MPI point-to-point latency and bandwidth. We can clearly see that the SHM channel shows much better performance than HCA channel on all message sizes. For instance, SHM channel outperforms HCA channel by up to 77% and 111% in terms of latency and bandwidth, respectively. CMA channel starts performing better than SHM channel after 8K message size. This is because, for large message size, single copy via CMA is faster than double copies via shared memory. While the system calls of CMA will incur too much overhead if we also use CMA for small message transfer. That is why we see CMA channel performs worse than SHM channel on small message sizes. Therefore, we use SHM channel for small message transfer. Further, Table I lists the count of message transfer operations during BFS of Graph 500 using each channel for different container deployment scenarios.

As we can see, neither native nor single container case is using HCA channel. When the message size is set to 8K, there are predominant 960,450 message transfer operations going through CMA channel. However, it introduces a large number of transfer operations on HCA channel once deploying more containers on a single host. For example, there are 376,071 HCA transfer operations in 2-Containers case, and the number increases to 791,341 on 4-Containers case. HCA channel communication turns out to be predominant in these two cases. Given that significant performance degradation of HCA channel, as shown in Figures 3(b) and 3(c), the performance bottleneck lies in the intra-host inter-container communication of MPI runtime, which goes through HCA channel and introduces the significant time delay.

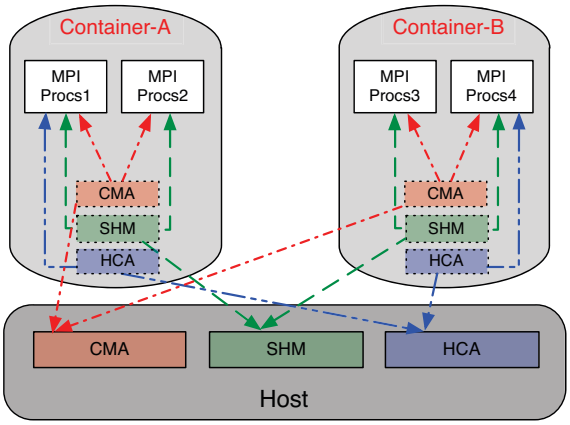


Figure 4: MPI Communication Channels for Container-based Cloud

IV. Design

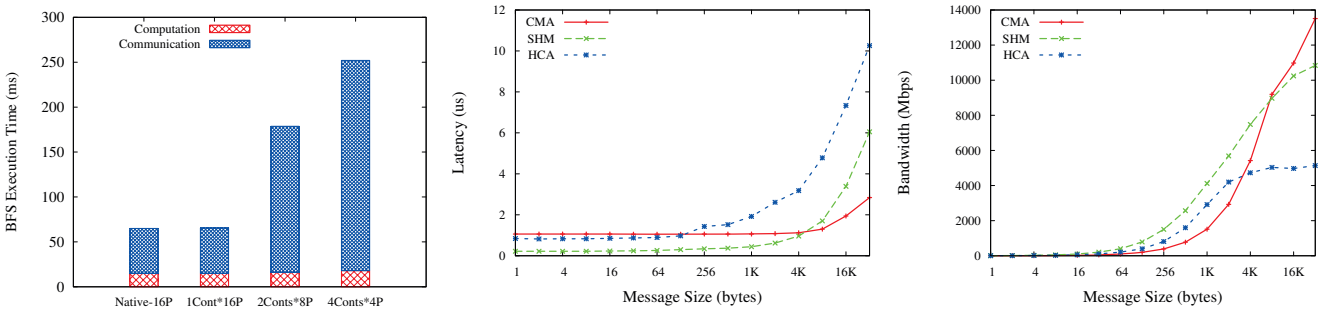
To address such performance bottleneck, we propose our design of a high performance locality-aware MPI library in this section. We further optimize the communication channels in the proposed MPI library.

A. Design Overview

Our design is based on MVAPICH2, an open-source MPI library over InfiniBand. For portability reasons, it follows a layered approach, as shown in Figure 5. The Abstract Device Interface V3 (ADI3) layer implements all MPI-level primitives. Multiple communication channels provide basic message delivery functionalities on top of communication device APIs. By default, there are three types of communication channels available in MVAPICH2: the shared memory channel communicating over user space shared memory to peers hosted on the same host; The CMA channel uses dedicated system calls to do the intra-node communication by directly reading/writing from/to another processes' address space, after the message size exceeds the predefined threshold; And the HCA channel communicating over InfiniBand user-level APIs to other peers.

Without any modification, default MVAPICH2 can run in container based virtualization environment. However, the SHM and CMA channels cannot be used across the different containers running on the same host for communication due to the different types of namespace isolation, which leads to performance limitations, as we show in Figures 3(b) and 3(c). Although sharing host's IPC and PID namespaces among containers provides the necessary conditions for SHM and CMA based communication across co-resident containers, the MPI communication still has to go through HCA channel. That is because each container has a unique hostname, default MPI runtime is not able to identify the co-residence of containers based on the different hostnames.

Therefore, we propose a high performance locality-aware MPI library to address such limitation. As shown in Figure 5,



(a) Graph 500 BFS Execution Time Breakdown (b) MPI Point-to-Point Latency Comparison (c) MPI Point-to-Point Bandwidth Comparison

Figure 3: Graph 500 BFS Execution Time Breakdown and Communication Channel Performance Comparison

we add a component 'Container Locality Detector' between ADI3 layer and channel layer. The Container Locality Detector is responsible for dynamically detecting and maintaining the information of local containers on the same host. According to the locality information, the communication path is rescheduled, so that the co-located containers can communicate through either SHM or CMA channel with better performance.

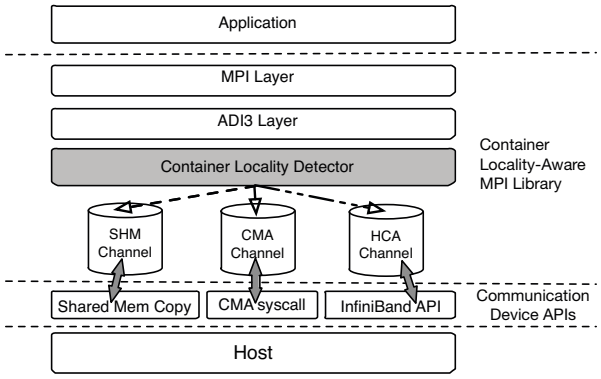


Figure 5: MVAPICH2 Stack Running in Container-based Environments

B. Container Locality Detector

As discussed in Section II-A, containers can share host's shared memory segments, semaphores, and message queues by sharing IPC namespace. Therefore, we create a container list structure on the shared memory region of each host, like /dev/shm/locality in Figure 6. During the initialization, each MPI process will write its own membership information into this shared container list structure according to its global rank. Figure 6 illustrates a scenario of launching an 8-process MPI job. The container A, B, and C are on the same host (e.g. host1). The MPI rank 0 and 1 are running in container-A, the rank 4 and 5 are running in container-B and container-C, respectively. While the other 4 ranks run on another host (e.g. host2). Then the three containers (ranks 0, 1, 4 and 5) will write their own membership information into positions 0, 1, 4 and 5 of the container list on host1 correspondingly. Other positions will be left blank. Similarly, other four MPI processes write at positions 2, 3, 6 and 7 of the container list on host2. Once the membership update of all processes completes, the real communication can take place subsequently. In this case, the local number of processes on host1 can be acquired by checking and counting whether the membership information has been written or not. Their local ordering can be maintained by their positions in the container list. Therefore, the written membership information on the container list indicates that they are co-resident.

In our proposed design, the container list is designed by using multiple bytes, as the byte is the smallest granularity of memory access without the lock. Each byte will be used to tag each container. This guarantees that multiple containers on the same host are able to write membership information on their corresponding positions concurrently without introducing lock&unlock operations. This approach reduces the overhead of locality detection procedure. Moreover, the proposed approach will not introduce much overhead of traversing the container list. Taking a one million processes MPI job, for instance, the whole container list only occupies 1 MB memory space. Therefore, it brings good scalability on virtualized MPI environment.

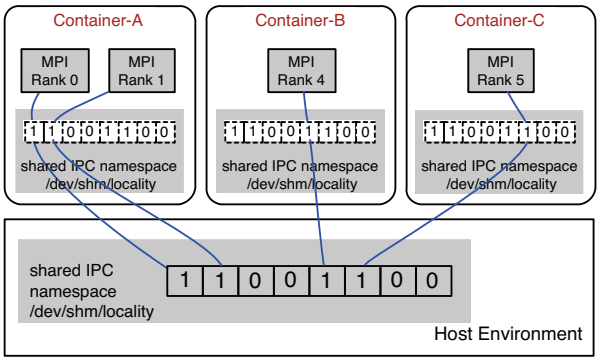


Figure 6: Container Locality Detection

C. Optimizing SHM and CMA Channels

The default setting in MVAPICH2 library has been optimized for the native environment, whereas it may not be the best setting for MPI communication over SHM and CMA channels in the container-based environment. Therefore, we need to optimize these two channels further in order to achieve high performance message passing for intra-host inter-container communication. For the SHM channel, there are four most important parameters for the eager and rendezvous protocols. However, with our proposed design to enable the CMA channel, messages transferred over the rendezvous protocol will directly go through the CMA channel. So we only need to consider two parameters for the eager protocol over SHM channel. In addition, we just show the optimization results on bandwidth and message rate, since there is no clear performance difference in terms of latency.

SMP_EAGER_SIZE defines the switch point between eager and rendezvous protocol. The eager protocol will go through SHM channel while the rendezvous protocol goes through CMA channel. As shown in Figure 7(a), we evaluate the performance impact of the different size of an eager message. The results indicate that the optimal performance can be achieved by setting SMP_EAGER_SIZE to 8K.

SMPI_LENGTH_QUEUE defines the size of shared buffer between every two processes on the same node for transferring messages smaller than SMP_EAGER_SIZE. Figure 7(b) shows the performance impact of the different size of length queue. We can see that the length queue with size 128K delivers the optimal performance.

D. Optimizing Communication for HCA Channel

Similarly, we need to optimize the HCA channel in container-based environments. **MV2_IBA_EAGER_THRESHOLD** specifies the switch point between eager and rendezvous protocol. If the threshold is too small, then it could incur additional overhead of RTS/CTS exchange during rendezvous transfer between sender and receiver for many message sizes. If it is too large, then it will require a larger amount of memory space for the library. Therefore, we need to keep the optimal threshold for inter-host inter-container communication. We measure the performance by setting MV2_IBA_EAGER_THRESHOLD to different values from 13K to 19K, as we can see in Figure 7(c). The results show that the HCA can deliver the optimal performance when this threshold is set to 17K for container environments.

V. Performance Evaluation

A. Experiment Setup

We use 16 bare metal InfiniBand nodes on **Chameleon Cloud** as our testbed, where each node has 24-core 2.3 GHz Intel Xeon E5-2670 processors with 128 GB main memory and equipped with Mellanox ConnectX-3 FDR (56 Gbps) HCAs with PCI Express Gen3 interfaces. We use CentOS Linux release 7.1.1503 (Core) with kernel 3.10.0-229.el7.x86_64 as the host OS and MLNX_OFED_LINUX-3.0-1.0.1 as the HCA driver. Docker 1.8.2 is deployed as the engine to build and run Docker containers. The privileged option is enabled to give the container access to the host HCA. All containers are set to share the host's PID and IPC namespaces.

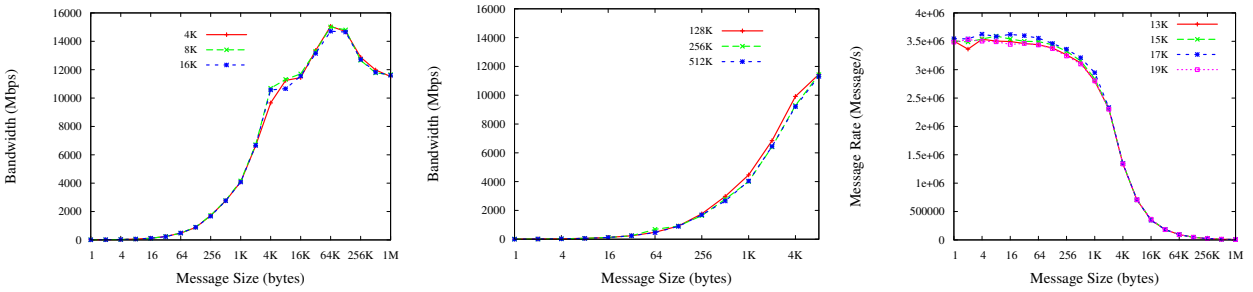
All applications and libraries used in this study are compiled with GCC 4.8.3 compiler. The MPI communication performance experiments use MVAPICH2-2.2b and OSU micro-benchmarks v5.0. Experimental results are averaged across multiple runs to ensure a fair comparison.

B. Point-to-Point Performance

In this section, we evaluate the point-to-point communication performance between two containers on a single host. The two containers are deployed on the same socket and the different socket to represent intra-socket and inter-socket cases. Figures 8(a)-8(c) show the 2-sided point-to-point communication performance in terms of latency, bandwidth and bi-directional bandwidth. The evaluation results indicate that compared to the default performance (Cont-*-Def), our proposed design (Cont-*-Opt) can significantly improve the point-to-point performance in both intra-socket and inter-socket cases. The performance benefits can be up to 79%, 191% and 407% for latency, bandwidth, and bi-directional bandwidth, respectively. If we compare the performance of our design with that of native MPI, we can see that our design only has minor overhead, which is much smaller than the overhead of default performance. For example, at 1KB message size, the MPI intra-socket point-to-point latency of default case is around 2.26μs, while the latencies of our design and native mode are 0.47μs and 0.44μs, respectively. In this case, our design only shows about 7% overhead. Figures 9(a)-9(f) present the 1-sided communication performance. The evaluation results show that compared with default performance, our proposed design brings up to 95% and 9X improvement in terms of latency and bandwidth for both put and get operations. Compared with native performance, there is also minor overhead with our proposed design. Taking put-bw for instance, at 4B message size, the intra-socket bandwidth of default case is 15.73Mbps, while our design and native mode achieve 147.99Mbps and 155.47Mbps, respectively. Through this comparison, we can clearly observe the performance benefits by optimizing MPI library with locality-aware design on container-based HPC cloud.

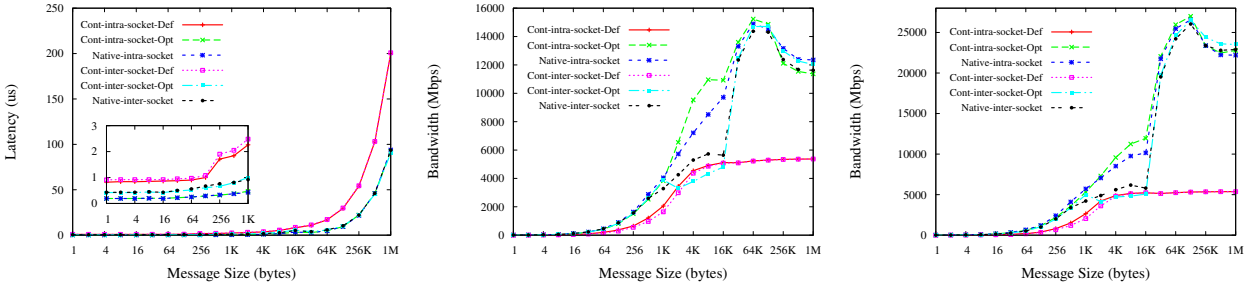
C. Collective Performance

In this section, we deploy 64 containers across 16 nodes evenly. By pinning different containers to different cores, we can avoid application competing for the same core and related performance degradation. Figures 10(a)-10(d) show the performance of broadcast, allreduce, allgather and alltoall



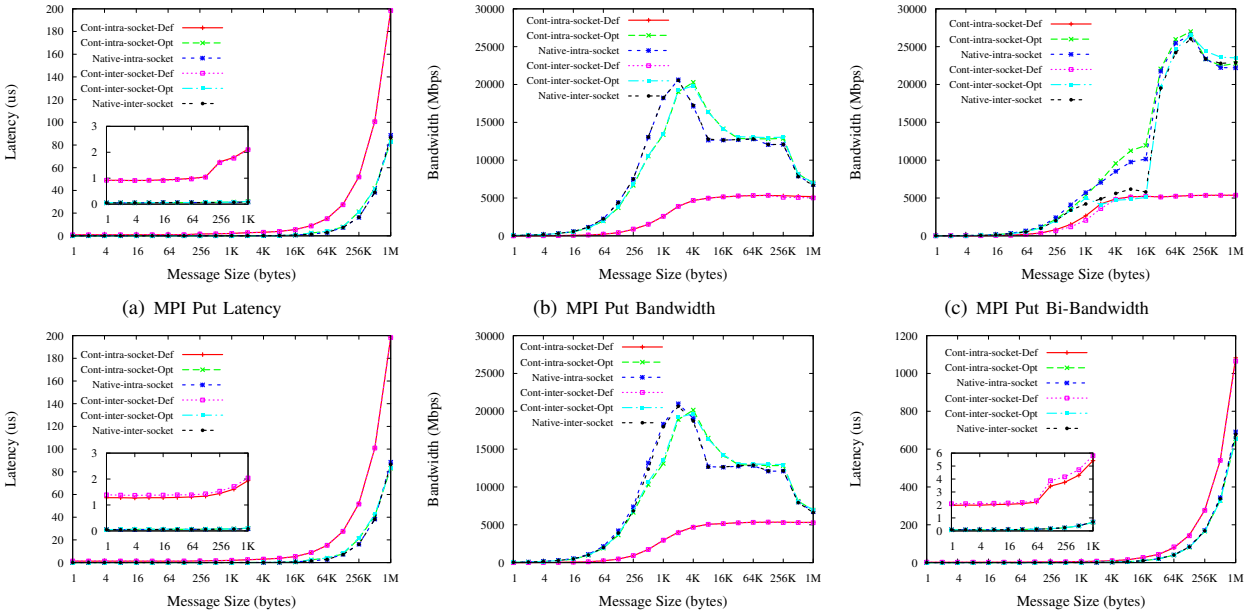
(a) Impact of CMA Threshold on Bandwidth (b) Impact of Length Queue for Small Message (c) Impact of Eager Threshold on Message Rate

Figure 7: Communication Channel Optimization



(a) MPI Point-to-Point Latency (b) MPI Point-to-Point Bandwidth (c) MPI Point-to-Point Bi-Bandwidth

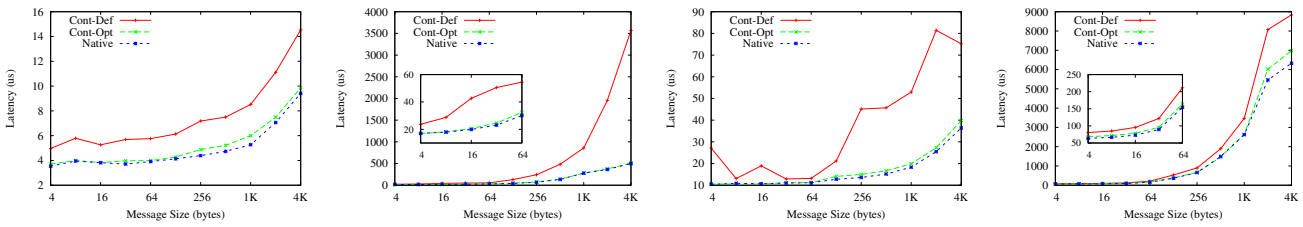
Figure 8: MPI Two-Sided Point-to-Point Communication Performance



(a) MPI Put Latency (b) MPI Put Bandwidth (c) MPI Put Bi-Bandwidth

(d) MPI Get Latency (e) MPI Get Bandwidth (f) MPI Get Accumulate Latency

Figure 9: MPI One-Sided Point-to-Point Communication Performance



(a) Broadcast (b) Allgather (c) Allreduce (d) Alltoall

Figure 10: Collective Communication Performance with 256 Processes

operations with 256 processes, respectively. With the proposed design, the communication across the four co-resident containers on each host can go through SHM and CMA channels, which improves the overall performance of collective operations. The evaluation results indicate that compared with default performance, our proposed design can clearly improve the performance by up to 59%, 64%, 86% and 28% for `MPI_Bcast`, `MPI_Allreduce`, `MPI_Allgather`, `MPI_Alltoall`, respectively. As inter-node transfer contributes a certain proportion of total amount of communication, we do not see the same benefits as that on point-to-point communication. In the meantime, the proposed design just incurs up to 9% overhead for the above four collective operations, compared with native performance.

D. Application Performance

In this section, we evaluate the performance of our proposed design with two end applications: Graph 500 and NAS Parallel Benchmarks (NPB). First, we re-run the experiments in Section I-A. As can be seen in Figure 11, the BFS execution time of our proposed design remains similar across the different container deployment scenarios. The

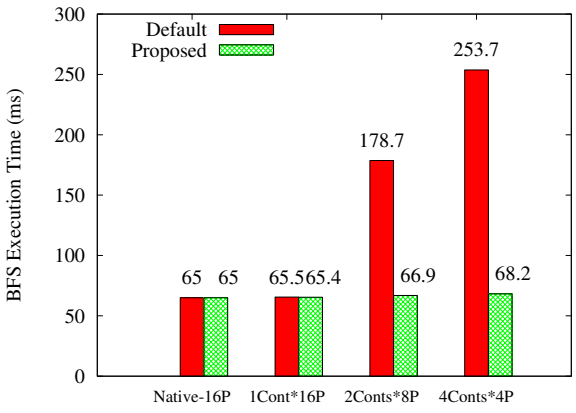


Figure 11: Execution Time of Graph 500 with 16 Processes using Default and Proposed MPI Library under Different Container Deployment Scenarios

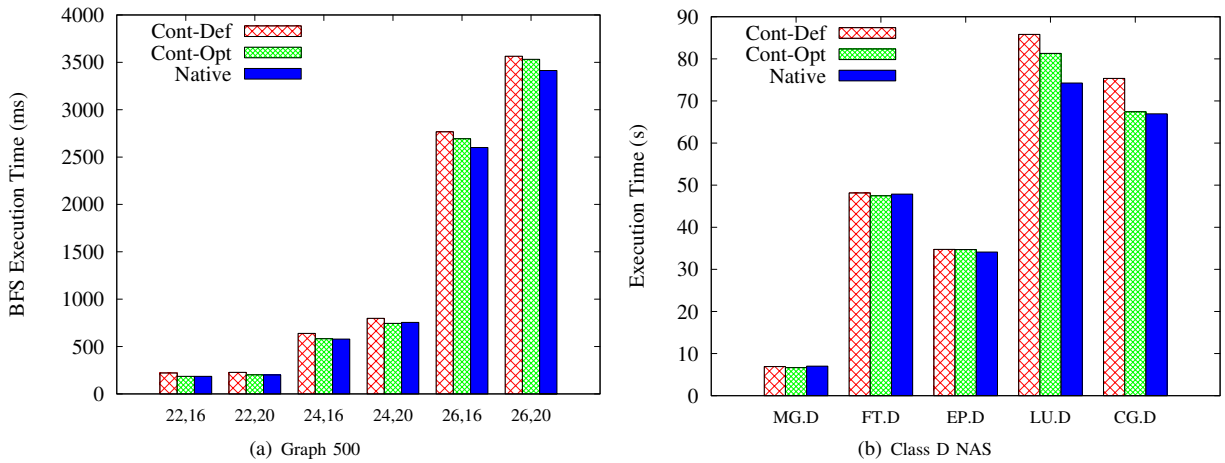
results verify that our proposed design is able to eliminate the performance bottleneck as we identified in Section III. Further, we run Graph 500 and Class D NAS with 256 processes across 16 nodes. The evaluation results show that compared with default case, the proposed design can reduce up to 16% (22,16) and 11% (CG) of execution time for Graph 500 and NAS, respectively. Compared with the native performance, the proposed design only has up to 5% and 9% overhead.

VI. Related Work

Several research efforts have exploited the facility of the inter-VM shared memory provided by the Xen hypervisor. XenSocket [23] is a one-way communications pipe for high-throughput inter-VM communications. It provides an interface to the underlying inter-VM shared memory communication mechanism by defining a new socket type and associated system calls. User-level applications and libraries need to be modified to explicitly invoke these calls. It lacks the support for automatic discovery and migration. XWay [24] intercepts TCP socket calls under the socket layer to provide transparent inter-VM communication for TCP oriented applications. It requires modifications to the implementation of network protocol stack in the core OS. XenLoop [25], as a kernel module, is inserted in the network protocol stack to intercept every outgoing packet from the network layer. It does not sacrifice user-level transparency and yet achieves high communication performance for co-resident VMs through shared memory channel. The transparent VM migration is also supported. IVC [26] is a user-level communication library intended for HPC applications that provide shared memory communication across co-resident VMs. VM migration is supported, though not fully transparently in user space. A VM-aware MPI library, MVAPICH2-ivc is further proposed based on IVC to benefit message passing HPC applications.

Our initial study of the performance characteristics of using SR-IOV with InfiniBand on KVM has shown that while SR-IOV enables low-latency communication [27], the MPI libraries need to be designed carefully and offer advanced features for improving intra-node inter-VM communication [28, 29]. Based on these studies, we provide an efficient approach to building HPC cloud with OpenStack over SR-IOV enabled InfiniBand clusters [30].

As a lightweight alternative, container technology has



(a) Graph 500 (b) Class D NAS

Figure 12: Application Performance with 256 Processes

been popularized during the last several years. More and more studies focus on evaluating the performance of different hypervisor-based and container-based solutions for HPC. Xavier et al. [31] conducted an in-depth performance evaluation of container-based virtualization (Linux VServer, OpenVZ, and LXC) and hypervisor-based virtualization (Xen) for HPC in terms of computing, memory, disk, network, application overhead and isolation. Wes Felter et al. [14] explored the performance of traditional virtual machine deployments (KVM) and contrasted them with the use of Docker. They used a suite of workloads that stress CPU, memory, storage, and networking resources. Their results show that containers result in equal or better performance than VMs in almost all cases. In addition, they found that both VMs and containers require tuning to support I/O-intensive applications [14]. Cristian et al. [16], evaluated the performance of Linux-based container solutions using the NAS parallel benchmarks, in various ways of container deployment. The evaluation shows the limits of using containers, the type of applications that suffer the most and until which level of oversubscription containers can deal with without impacting considerably the application performance. Yuyu et al. [15] compare the virtualization (KVM) and containerization (Docker) techniques for HPC in terms of features and performance using up to 64 nodes on Chameleon testbed with 10GigE networks.

Instead of performance characterization, we focus on building efficient container-based HPC cloud under different container deployment scenarios in this paper. We identify and address a clear performance bottleneck for MPI applications running in multi-container per host environments by proposing a high performance locality-aware MPI library.

VII. Conclusion and Future Work

In this paper, we identified the performance bottleneck for MPI application running in the container-based HPC cloud by executing a detailed performance analysis. To eliminate this bottleneck, we proposed a high performance locality-aware MPI library for container-based HPC cloud. By the help of locality-aware design, MPI library is able to dynamically and efficiently detect co-resident containers at runtime, so that shared memory and CMA based communication can be executed to improve the communication performance across the co-resident containers. We further analyzed and optimized core mechanisms and design parameters of MPI library for SHM, CMA and HCA channels in container-based HPC cloud.

Through a comprehensive performance evaluation, our proposed locality-aware design can significantly improve the communication performance across co-resident containers. The evaluation results indicate that compared with default case, our proposed design can bring up to 95% and 9X performance improvement for MPI point-to-point communication in terms of latency and bandwidth. On the aspect of collective operation, the proposed design can achieve up to 59%, 64%, 86% and 28% improvement for MPI broadcast, allreduce, allgather and alltoall operations, respectively. The evaluation results on application level demonstrate that the proposed design can reduce up to 16% and 11% of execution time for Graph 500 and NAS across 64 containers, respectively. In the meantime, our results also show that the proposed locality-aware design can deliver near-native performance for applications with less than 5% overhead. Therefore, the proposed high performance locality-aware MPI library reveals significant potential to be utilized to efficiently build large scale container-based HPC cloud. For future work, we plan on exploring the performance characterization of other programming models (e.g. PGAS) in container-based HPC cloud. We also plan to make this design available through public release of MVAPICH2-Virt library.

REFERENCES

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

[2] VMware ESX/ESXi, https://www.vmware.com/products/esxi-and-esx/overview.

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

[4] "Amazon EC2," http://aws.amazon.com/ec2/.

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

[6] VMware vCloud Air, http://vcloud.vmware.com/.

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

[8] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High Performance Network Virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, 2012.

[9] Z. Huang, R. Ma, J. Li, Z. Chang, and H. Guan, "Adaptive and Scalable Optimizations for High Performance SR-IOV," in *Proceeding of 2012 IEEE International Conference Cluster Computing (CLUSTER)*. IEEE, 2012, pp. 459–467.

[10] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, 2007.

[11] Linux VServer, http://linux-vserver.org.

[12] Linux Containers, https://linuxcontainers.org.

[13] Docker, https://www.docker.com/.

[14] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," Tech. Rep. RC25482 (AUS1407-001), 2014.

[15] Y. Zhou, B. Subramaniam, K. Keahey, and J. Lange, "Comparison of Virtualization and Containerization Techniques for High Performance Computing," in *Proceedings of the 2015 ACM/IEEE conference on Supercomputing*, Austin, USA, Nov 2015.

[16] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance Evaluation of Containers for HPC," in *10th Workshop on Virtualization in High-Performance Cloud Computing (VHPC)*, Vienna, Austria, Aug 2015.

[17] The Graph500, http://www.graph500.org.

[18] B. P. C. M. Andreea Anghel, German Rodriguez and G. Dittmann, "Quantifying Communication in Graph Analytics," ser. International Supercomputing Conference (ISC), 2015.

[19] Cross Memory Attach (CMA), http://kernelnewbies.org/Linuxi_3.2.

[20] Chameleon Cloud, https://www.chameleoncloud.org/.

[21] Infiniband Trade Association, http://www.infinibandta.org.

[22] mpiP, http://mpip.sourceforge.net/.

[23] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "XenSocket: A High-throughput Interdomain Transport for Virtual Machines," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware (Middleware)*, Newport Beach, USA, 2007.

[24] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Seattle, USA, 2008.

[25] J. Wang, K.-L. Wright, and K. Gopalan, "XenLoop: A Transparent High Performance Inter-vm Network Loopback," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC)*, Boston, USA, 2008.

[26] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda, "Virtual Machine Aware Communication Libraries for High Performance Computing," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, Reno, USA, 2007.

[27] J. Jose, M. Li, X. Lu, K. Kandalla, M. Arnold, and D. Panda, "SR-IOV Support for Virtualization on InfiniBand Clusters: Early Experience," in *Proceedings of 13th IEEE/ACM International Symposium Cluster, Cloud and Grid Computing (CCGrid)*, May 2013, pp. 385–392.

[28] J. Zhang, X. Lu, J. Jose, R. Shi, D. K. Panda, "Can Inter-VM Shmem Benefit MPI Applications on SR-IOV based Virtualized InfiniBand Clusters?" in *Proceedings of 20th International Conference Euro-Par 2014 Parallel Processing*, Porto, Portugal, August 25-29 2014.

[29] J. Zhang, X. Lu, J. Jose, M. Li, R. Shi, D. K. Panda, "High Performance MPI Library over SR-IOV Enabled InfiniBand Clusters," in *Proceedings of International Conference on High Performance Computing (HiPC)*, Goa, India, December 17-20 2014.

[30] J. Zhang, X. Lu, M. Arnold, and D. K. Panda, "MVAPICH2 over OpenStack with SR-IOV: An Efficient Approach to Build HPC Clouds," in *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2015, pp. 71–80.

[31] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Belfast, Northern Ireland, Feb 2013, pp. 233–240.