Cluster Computing
https://doi.org/10.1007/s10586-021-03460-8

Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study

Peini Liu[1,2] · Jordi Guitart[1,2]

Received: 15 February 2021 / Revised: 22 October 2021 / Accepted: 27 October 2021
© The Author(s) 2021

Abstract
Containerization technology offers an appealing alternative for encapsulating and operating applications (and all their dependencies) without being constrained by the performance penalties of using Virtual Machines and, as a result, has got the interest of the High-Performance Computing (HPC) community to obtain fast, customized, portable, flexible, and reproducible deployments of their workloads. Previous work on this area has demonstrated that containerized HPC applications can exploit InfiniBand networks, but has ignored the potential of multi-container deployments which partition the processes that belong to each application into multiple containers in each host. Partitioning HPC applications has demonstrated to be useful when using virtual machines by constraining them to a single NUMA (Non-Uniform Memory Access) domain. This paper conducts a systematical study on the performance of multi-container deployments with different network fabrics and protocols, focusing especially on Infiniband networks. We analyze the impact of container granularity and its potential to exploit processor and memory affinity to improve applications' performance. Our results show that default Singularity can achieve near bare-metal performance but does not support fine-grain multi-container deployments. Docker and Singularity-instance have similar behavior in terms of the performance of deployment schemes with different container granularity and affinity. This behavior differs for the several network fabrics and protocols, and depends as well on the application communication patterns and the message size. Moreover, deployments on Infiniband are also more impacted by the computation and memory allocation, and because of that, they can exploit the affinity better.

Keywords Performance · Containerization · InfiniBand · Multi-container · Singularity · Docker

1 Introduction

Following the trend of Cloud computing, the HPC community has also started to adopt containerization instead of hardware virtualization to benefit from some of its well-known advantages [1], such as the encapsulation of specific software environments for each user, which allows for customization, portability, and research reproducibility; the isolation of users from the underlying system and from other users, which allows for security and fault protection; and the agile and fine-grain resource allocation and balancing, which allows for efficient cluster utilization and failure recovery [2, 3].

A matter of the utmost importance for HPC users is that the containers running their applications can leverage the underlying HPC resources such as Infiniband networks, which offer high-speed networking capabilities with improved throughput and low latency through the use of Remote Direct Memory Access (RDMA)[4].

Previous work on this area has demonstrated that containerized HPC applications can exploit InfiniBand networks, especially when they run on a single container per host that shares the host network namespace. Whereas some works have evaluated more sophisticated networking modes, such as overlay networks, they have just superficially considered multi-container deployments which partition the processes that belong to each application into

✉ Peini Liu
peini.liu@bsc.es

Jordi Guitart
jordi.guitart@bsc.es

1 Computer Science Department, Barcelona Supercomputing Center (BSC), Barcelona, Spain

2 Computer Architecture Department, Universitat Politecnica de Catalunya (UPC), Barcelona, Spain

multiple containers in each host. Partitioning HPC applications has demonstrated to be useful when using virtual machines by constraining them to a single NUMA (Non-Uniform Memory Access) domain [5], and can also increase the utilization of the hosts since small-sized tasks can be packed more easily. Consequently, it is essential to understand the performance implications of multi-container deployment schemes for HPC workloads on Infiniband clusters, focusing especially on understanding how the container granularity and its combination with processor and memory affinity impact the performance when using different networking modes.

Performance analysis of HPC applications in containerized environments is an ongoing research problem [6, 7]. Most related works evaluate single-container deployments and emphasize the possibility that deploying an HPC workload into a single container can achieve native performance [8, 9], while others use orchestrators to manage the container placement at scale and study the incurred overheads [2, 7, 10]. Few works include experiments with different container granularity [7, 11], but none of them provide a deep understanding of the impact of such multi-container deployments on the performance of HPC workloads, which considers different containerization technologies, container grain sizes, and includes processor and memory affinity. The ability to provision InfiniBand to Docker and Singularity containers has been shown in [4, 7, 12, 13]. However, it is still unclear how multi-container deployment schemes with different affinity settings perform with various network interconnects and protocols, and how different communication patterns and message sizes impact the performance of containerized HPC workloads.

In this paper, we present a detailed performance characterization of different containerization technologies (including Docker and Singularity) for HPC workloads on InfiniBand clusters through different dimensions, namely network interconnects (including Ethernet and InfiniBand) and protocols (including TCP/IP and RDMA), networking modes (including host, MACVLAN, and overlay networking), and processor and memory affinity. We aim to answer some research questions including: i) What is the performance of different containerization technologies with various network interconnects and protocols? ii) What is the impact of container granularity on multi-container deployment scenarios using different network interconnects and protocols? iii) What is the impact of processor and memory affinity on multi-container deployment scenarios using different network interconnects and protocols?

2 Background

2.1 Containerization

2.1.1 Docker

Docker,[1] the most popular containerization technology, builds upon resource isolation and limitation features of the Linux kernel, such as `namespaces` and `cgroups`, respectively. Also, it adds a union-capable file system such as OverlayFS. Without the hypervisor needed for virtual machines, Docker contains a lightweight engine to control and manage its containers. Docker also allows containers to share the underlying host kernel, including the libraries, modules, kernel functions, and a root file system. Regarding runtime isolation, Docker containers are defined into some operational spaces (e.g., Network, PIDs, UIDs, IPC) implemented by means of `namespaces`. Regarding resource limitation, some sets of dedicated resources defined through `cgroups` can be allocated to the Docker containers.

2.1.2 Singularity

Singularity[2] containers are mostly used in HPC environments where they are proven to introduce less overhead than Docker while providing more reliable security guarantees [6]. Regarding security, Singularity does not create containers as spawned child processes of a root-owned daemon. Regarding performance, Singularity enables all the containers to use the underlying HPC environment naturally (without namespaces isolation). Because of this feature, the integration between Singularity and MPI can be transparent to the user. These make Singularity a first-class choice for HPC and scientific simulations [14]. In late 2018, Singularity 3.0 was released [15]. This version brings a new functionality (so-called instances) to run containers in "daemon" mode, which allows running containers as services in the background. Singularity instances can have isolated network resources, and they also support cgroups functionality to restrict resource usage. MPI applications can run in Singularity instances as if they were running in separated hosts, having their own network identity and using an SSH backend service to communicate.

2.2 Multi-host container networking

Containers could communicate across hosts through both underlay and overlay networking approaches. In underlay network approaches, containers are directly exposed to the

[1] https://www.docker.com/.

[2] https://sylabs.io/.

host network. When running a single container per host, the container could run in host mode and share the network stack and namespace of the host. When running one or multiple containers per host, we also consider MACVLAN as an underlay network approach. MACVLAN allows configuring multiple MAC addresses on a single physical interface. This can be used to assign a different MAC address (and consequently a different IP address) to each container, making it appear to be directly connected to the physical network. In that way, containers can be accessed through their IP addresses. However, MACVLAN requires those addresses to be on the same broadcast domain as the physical interface. MACVLAN is a simple and efficient approach but the underlying network could restrict its application, in particular by limiting the number of different MAC addresses on a physical port or the total number of MAC addresses supported, or forbidding multiple MAC addresses to be assigned on a single physical interface. Furthermore, MACVLAN is not generally supported for wireless network interfaces.

In overlay network approaches, a logical network between the containers is built using networking tunnels to deliver communication across hosts. Those tunnels add an additional level of encapsulation to the underlying network. Because of this, they may introduce some extra overhead when compared with an underlay approach, due to the encapsulation overhead of the frame size and the processing overhead on the server. Nevertheless, overlay network approaches are very flexible as they decouple the virtual network topology from the physical network, which supports for instance the mobility of components independently of the physical network. In addition, they essentially support an unlimited number of components, as they do not suffer from restrictions to the number of addresses imposed by the physical network [16].

2.3 InfiniBand interconnect

InfiniBand (IB) [17] interconnect can provide high throughput and low latency communication across systems for distributed and parallel applications. IB is well-known and supported by most operating systems and cluster vendors. IB comprises two channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCA provides the hardware visibility at the user-level for communication. OpenMPI follows the standard of the software stack from the OpenFabrics Alliance for the Remote Direct Memory Access (RDMA) through InfiniBand, which allows processes to access the memory of a remote node process without the CPU intervention [18].

InfiniBand interconnect can also support other communication protocols, for example, TCP/IP network protocol stack can be adapted for InfiniBand through TCP/IP over IB (IPoIB) [19]. IPoIB is a Linux kernel module that enables InfiniBand hardware devices to encapsulate IP packets into IB datagrams or connected transport services. When IPoIB is applied, an InfiniBand device is assigned an IP address and accessed just like any regular TCP/IP hardware device [18]. The IPoIB driver supports two modes of operation: datagram and connected. In datagram mode, the IB unreliable datagram transport is used. In connected model, the IB reliable connected transport is used.

3 Evaluation methodology

Our performance characterization will consider the four dimensions in Fig. 1, namely containerization technologies, networking modes, interconnects and protocols, and affinity, respectively.

3.1 Containerization technologies

In this dimension, we choose Docker, Singularity, and its variant with container instances (hereinafter called Singularity-instance and Singularity-instance + cgroup) as representative containerization technologies. The bare-metal performance is also provided to evaluate the corresponding overhead of each containerization technology.

3.2 Interconnects and protocols

We consider 1-Gigabit Ethernet and InfiniBand interconnects in this dimension. We evaluate the performance of the different containerization technologies configured with several networking modes to operate on these interconnects through various protocols, such as TCP/IP and RDMA. Details are as we described in Sect. 2.3.

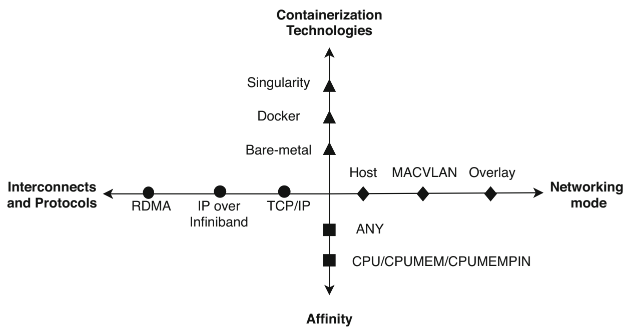


Fig. 1 Four evaluation dimensions

3.3 Networking mode

For Docker and Singularity-instance, the networking modes considered in the experiments depend on the number of deployed containers per host. As described in Sect. 2.2, when deploying a single container per host, we could use an underlay networking approach by sharing the host network with the containers or by setting a MACVLAN address to each container, or use an overlay networking approach through a network VXLAN tunnel that enables the communication across hosts. When deploying multiple containers per host, we can only use MACVLAN or overlay networking approaches for the communication of multiple containers across hosts.

For default Singularity, as the containers within the same host do not have isolated network namespaces (they run in the same network namespace as the host), they can share the host network.

3.4 Affinity settings

The affinity settings for our multi-containerized deployment scenarios include CPU, CPUMEM, and CPUMEMPIN, which are all compared to ANY. We assume a number of hosts N_h, where each has a number of containers N_{ctn}. Each container hosts a number of processes N_{mpi}, so that $N_{ctn} \times N_{mpi} = K$, which is kept constant in all the deployment scenarios (e.g., 128). The hardware platform provides a number of *CPU* cores and *MEM* nodes from one or more sockets $S = \{socket_s | s = 0, \ldots, N_{socket} - 1\}$, where each socket has P cores. Hence, for each application comprising a set of processes $MPI = \{mpi_j | j = 1, \ldots, N_{mpi}\}$ hosted in a set of containers $CTN = \{ctn_i | i = 1, \ldots, N_{ctn}\}$ which run on a set of hosts $HOST = \{host_h | h = 1, \ldots, N_h\}$, each affinity setting defines a mapping: $Map_{h,i,j} \rightarrow CPU_{h,s,x\rightarrow y} + MEM_{h,s}$ where h, s and $x \rightarrow y$ denote the assigned host, socket, and the range of cores, respectively. Each affinity setting works as follows:

(I) ANY: processes do not have any processor or memory affinity, they could access all the resources provided to this application, and the actual distribution is decided by the operating system. Thus, the mapping of ANY scenarios could be expressed as:

$$Map_{h,i,j} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{h,s,s\times P \rightarrow s\times P + \frac{N_{cpu}\times N_{ctn}}{N_{socket}} - 1} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_{h,s} \end{cases} \tag{1}$$

(II) CPU: we define a specific processor affinity for each container to a set of cores from two different sockets. The mapping of CPU scenarios could be formulated as follows:

$$Map_{h,i,j} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{h,s,s\times P+(i-1)\times\frac{N_{cpu}}{N_{socket}}} \\ {}_{\rightarrow s\times P+i\times\frac{N_{cpu}}{N_{socket}}-1} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_{h,s} \end{cases} \tag{2}$$

(III) CPUMEM: we define a specific processor and memory affinity for each container to a set of cores belonging to a single socket and to the corresponding local memory node. The mapping of CPUMEM scenarios could be calculated as follows, provided that the number of cores requested by each container is lower than the cores each socket provides.

$$Map_{h,i,j} \rightarrow \begin{cases} CPU_{h,\lceil\frac{i}{N_{cps}}\rceil-1,\left(\left\lceil\frac{i}{N_{cps}}\right\rceil-1\right)\times P+((i-1)} \\ {}_{-N_{cps}\times\left(\left\lceil\frac{i}{N_{cps}}\right\rceil-1\right))\times N_{cpu}} \\ {}_{\rightarrow\left(\left\lceil\frac{i}{N_{cps}}\right\rceil-1\right)\times P+(i-N_{cps}} \\ {}_{\times\left(\left\lceil\frac{i}{N_{cps}}\right\rceil-1\right))\times N_{cpu}-1} \\ MEM_{h,\left\lceil\frac{i}{N_{cps}}\right\rceil-1} \end{cases} \tag{3}$$

where N_{cps} refers to the number of containers per socket and is calculated as N_{ctn}/N_{socket}.

(IV) CPUMEMPIN: this scheme has the same setting as CPUMEM about the affinity of the containers, but it enables the 1-to-1 process-to-processor binding inside the container. Thus each process is mapped into a specific core:

$$Map_{h,i,j} \rightarrow \begin{cases} CPU_{h,\left\lceil\frac{i}{N_{cps}}\right\rceil-1,\left(\left\lceil\frac{i}{N_{cps}}\right\rceil-1\right)\times P+((i-1)} \\ {}_{-N_{cps}*\left(\left\lceil\frac{i}{N_{cps}}\right\rceil-1\right))\times N_{cpu}+j-1} \\ MEM_{h,\left\lceil\frac{i}{N_{cps}}\right\rceil-1} \end{cases} \tag{4}$$

4 Performance evaluation

In this section, we first describe our experimental setup. Then, we present the results when deploying a single container per host with different networking modes. Finally, we provide the results of multi-container deployments, where we evaluate the impact of container granularity and processor and memory affinity when using different network interconnects and protocols.

4.1 Experimental setup

4.1.1 Hardware

Our experiments are executed on a five-node HPC InfiniBand cluster. Each host consists of 2× Intel 2697v4 CPUs (18 cores each, hyperthreading disabled), 256 GB RAM, 60 TB GPFS file system, 1-Gigabit Ethernet network, and Mellanox Technologies MT27700 Family ConnectX-4 InfiniBand (EDR 100Gb/s Adapter), which works on datagram mode.

4.1.2 Software

For both hosts and containers, we use CentOS release 7.6.1810 with host kernel 3.10.0-957.27.2.el7.x86_64 and MLNX_OFED_LINUX-4.7-1.0.0.1 as the HCA driver. Docker 19.03.10 and Singularity 3.5.1 are used to conduct all the experiments. OpenMPI 4.0.3rc3 and all the benchmarks are compiled with gcc 5.5.0 compiler.

4.1.3 Benchmarks

(1) *OSU Benchmark* OSU Benchmark[3] is a suite of benchmarks that measure the MPI-level operation performance. We choose this benchmark for understanding MPI communication performance with different message sizes. We use version 5.6.3. (2) *HPCC Benchmark* The HPC Challenge benchmark suite[4] is widely used to evaluate the performance of HPC systems. Its design goal is to enable complete understandings of the performance characteristics of platforms [20]. It consists of several benchmarks that show the performance impact of real-world HPC applications. For example, the capability of processor floating point computation (e.g., DGEMM, FFT), memory bandwidth (e.g., STREAM, FFT) and latency (e.g., RandomAccess), and communication bandwidth (e.g., RandomRing Bandwidth, PTRANS, FFT) and latency (e.g., RandomAccess) [21, 22]. We use v1.5.0.

4.1.4 Networking mode and protocol settings

We evaluated various network interconnects and protocols, namely TCP/IP protocol on Ethernet, TCP/ IP protocol over InfiniBand (IPoIB), and RDMA natively on InfiniBand. Detailed network and protocol settings for each containerization technology are shown in Table 1. Single container per host scenarios are tested with three different networking modes: Host, MACVLAN, and Overlay. Note that MACVLAN does not work with InfiniBand, so we tested it only with TCP/IP on Ethernet. Multiple containers per host scenarios are tested only with the overlay networking mode, as this is the only mode that allows running multiple containers per host on all the network interconnects.

Docker implements its own networking specification called the Container Network Model,[5] which supports multi-host networking through both underlay (based on MACVLAN) and overlay native drivers. The overlay network for Docker used in our experiments is not using Docker Swarm but configuring an external etcd[6] discovery service. For Singularity-instance, it uses the CNI[7] plugins for defining various basic networks such as bridge, ipvlan or macvlan. We use the knowledge from our previous work to enable the interconnection between Singularity instances across hosts [16]. As for Singularity-instance + cgroups, we keep the same network settings as Singularity-instance but enabling the cgroup support by adding `apply-cgroups` parameter.

4.1.5 Granularity deployment scenarios

We study both single- and multi-container deployment schemes. One host acts as the master for launching the experiments and the other four hosts run each benchmark consisting of 128 processes in total. Detailed settings are shown in Table 2. For Docker, Singularity-instance, and Singularity-instance + cgroups, we generate scenarios SCE1–SCE6 by increasing the number of containers per host, in particular 1, 2, 4, 8, 16, and 32 containers per host, but decreasing the number of processes per container, that is, finer-grained container granularity (i.e., 32, 16, 8, 4, 2, and 1 processes per container, respectively).

4.1.6 Scheduling and binding policy

OpenMPI's default mapping and binding policy schedules in a round-robin fashion through slots and automatically binds processes to socket if the number of processes is more than two and binds processes to cores if the number of processes is less or equal than two. However, this binding policy is inadequate when enabling multi-container deployments because processes in different containers are not aware of their peers and always bind to the first socket by default. Thus, in experiments (Sects. 4.2 and 4.3), we

[3] https://mvapich.cse.ohio-state.edu/benchmarks/.

[4] http://icl.cs.utk.edu/hpcc/.

[5] https://github.com/docker/libnetwork/blob/master/docs/design.md.

[6] https://etcd.io.

[7] https://github.com/containernetworking/cni.

Table 1 Networking mode and protocol settings

Containerization	Networking mode	Protocols
Bare-metal(B)	Host	TCP/IP; IPoIB; RDMA
Docker(D)	Host Overlay MACVLAN	TCP/IP; IPoIB; RDMA TCP/IP; IPoIB; RDMA TCP/IP
Singularity- instance(SI)	Host Overlay MACVLAN	TCP/IP; IPoIB; RDMA TCP/IP; IPoIB; RDMA TCP/IP
Singularity(S)	Host	TCP/IP; IPoIB; RDMA

Table 2 Container granularity settings

Containerization	No. of containers per host (NC)	No. of processes per container (NP)
Docker (D)	1, 2, 4, 8, 16, 32	32, 16, 8, 4, 2, 1
Singularity-instance (SI)	1, 2, 4, 8, 16, 32	32, 16, 8, 4, 2, 1
Singularity-instance + cgroups (SI + CG)	1, 2, 4, 8, 16, 32	32, 16, 8, 4, 2, 1

use rankfiles with specific mappings between processes and cores to ensure a uniform distribution. For experiment (Sect. 4.4), the rankfiles are derived from the formulas presented in Sect. 3. In addition, in our experiments, we restrict the resources to Docker and Singularity-instance + cgroups containers by setting `cpuset-cpus` and `cpuset-mems` parameters and specifying cpus and mems options within the cgroup file used by `apply-cgroups`, respectively.

4.1.7 Performance analysis tools

We use Paraver[8] to profile MPI usage patterns of the benchmarks. We capture performance event counters and operating system metrics (through Perf[9]), such as context-switches, migrations, and memory accesses, from representative executions of the benchmarks and we use them to explain the obtained performance results.

4.2 Impact of containerization on a single container per host deployment scenario with different network fabrics

We use the MPI_Alltoallv Latency Test from the OSU benchmark suite to evaluate the global latency of ranks sending and receiving data. This test spreads 128 MPI processes across four hosts, and then all of them send data to and receive data from all the others. In addition, we use the OSU Bidirectional Bandwidth Test to measure the maximum aggregate bandwidth between two adjacent nodes that send out a fixed number of back-to-back messages between them. As both tests perform a large number of iterations and already provide an averaged result, we display the outcome of a single execution for each sample.

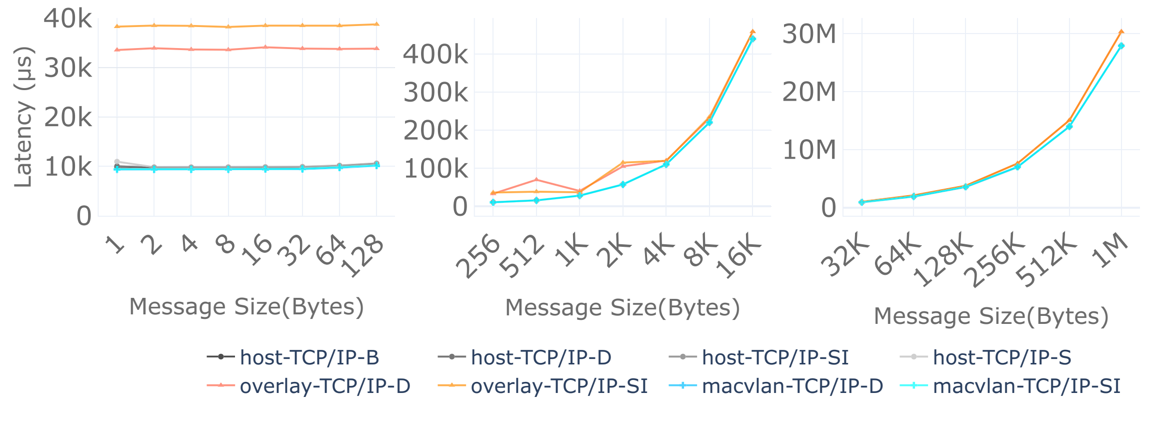
Figures 2, 3 and 4 show the performance of different containerization technologies with several network fabrics and protocols. As expected, the RDMA protocol has higher performance and lower latency than IPoIB, and those two perform better than TCP/IP in all the container networking modes.

Default Singularity reaches the same performance as bare-metal in all the scenarios, given that running on default Singularity is equivalent to running processes on bare-metal, as all the container processes on a given host reside in the same namespaces (network, IPC, etc.) as the host.

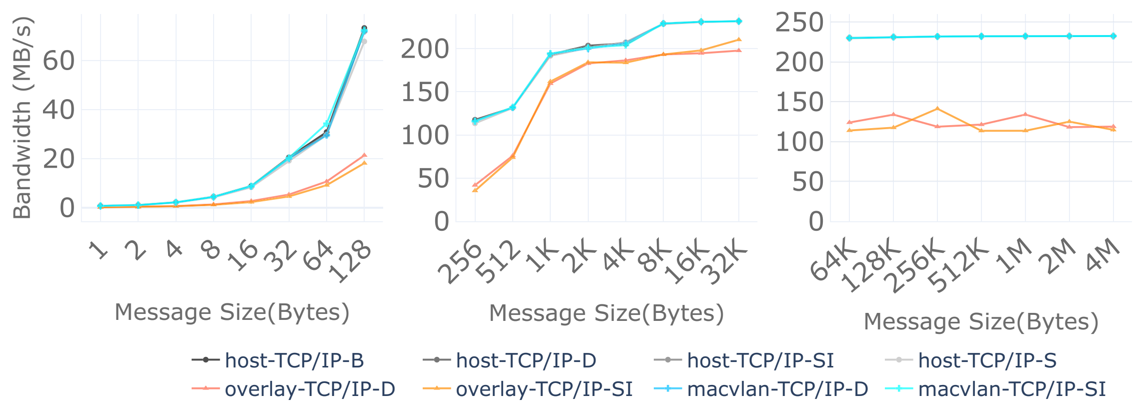
For Docker and Singularity-instance, underlay container networking approaches, such as host networking and MACVLAN networking, also achieve comparable performance to bare-metal experiments. With host networking, the single container shares the same network namespace as the host. With MACVLAN networking, a container gets unique MAC and IP addresses and is exposed directly to the underlay network. In contrast, overlay networking brings explicit latency increase and bandwidth degradation for Docker and Singularity-instance. This occurs because all the communications among containers must be encapsulated through a tunnel, and this additional encapsulation incurs overhead (i.e., reduces the amount of application data sent on each network packet). For TCP/IP over Ethernet, latency increments are more significant with small messages, whereas bandwidth degradation occurs for all message sizes. For example, Docker overlay networking shows 244% (8B), 9% (1 MB) latency increase and 70% (8B), 49% (1 MB) bandwidth degradation compared to bare-metal. For IPoIB, overlay networking shows

[8] https://tools.bsc.es/paraver.

[9] http://man7.org/linux/man-pages/man1/perf.1.html.



(a) OSU MPI_Alltoallv Latency



(b) OSU Bidirectional Bandwidth

Fig. 2 TCP/IP over Ethernet: Latency (**a**) and bandwidth (**b**) for scenario SCE1 with different networking modes

significant latency increments (especially with large messages) and bandwidth degradation with all sizes compared to bare-metal. In particular, Docker presents 26% (8B), 211% (1 MB) latency increase and 70% (8B), 74% (1 MB) degradation on bandwidth. Both TCP/IP and IPoIB can benefit from an increment of the MTU (Maximum Transmission Unit) value to attenuate the incurred overhead by overlay networking for communication-intensive workloads.

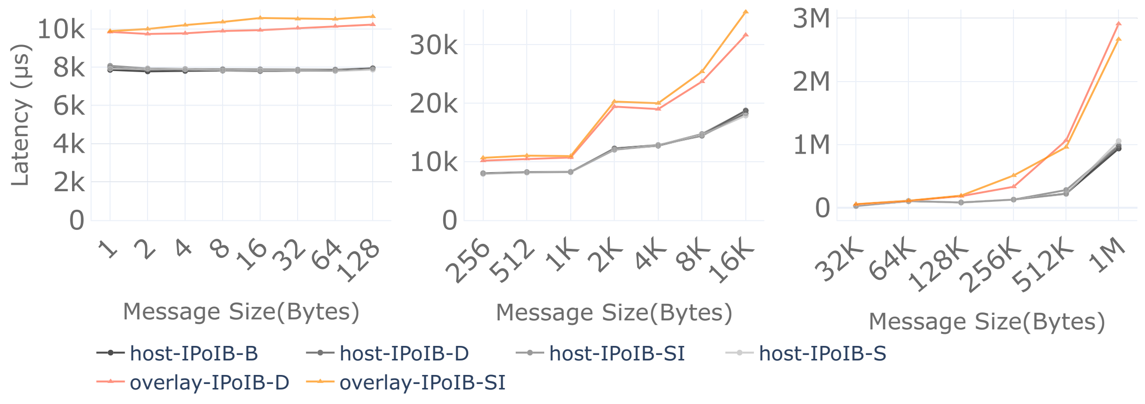
On the other side, overlay networking on RDMA over InfiniBand has negligible performance degradation for all the containerization technologies on bandwidth and latency regarding the bare-metal baseline. This is because the data communications among processes are performed through RDMA and the overlay network connection is only used for initiating and setting up the nodes.

4.3 Impact of container granularity on multi-container per host deployment scenarios with different network fabrics

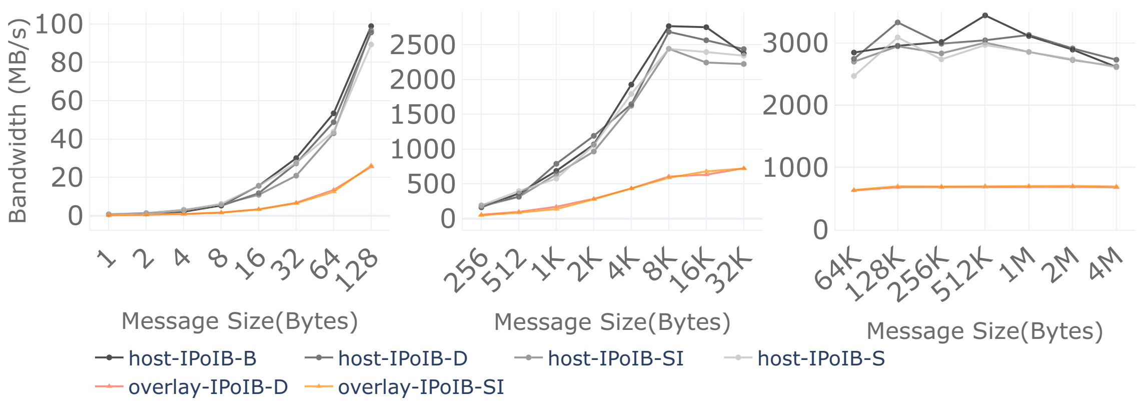
In this section, we evaluate the impact of container granularity on multi-container deployments. First, we use again the MPI_Alltoallv Latency Test from the OSU benchmark through different message sizes. Then, we use the HPCC benchmark suite to assess how different MPI communication patterns are impacted by container granularity. HPCC results are derived from the average of ten executions, and we plot the median value and the standard deviation after eliminating outliers that lie beyond 1.5 times the interquartile range. As justified before, all the multi-container experiments use overlay networks.

4.3.1 OSU MPI_Alltoallv latency

Figures 5, 6 and 7 show the MPI_Alltoallv latency of multi-container deployments for Docker and Singularity-



(a) OSU MPI_Alltoallv Latency



(b) OSU Bidirectional Bandwidth

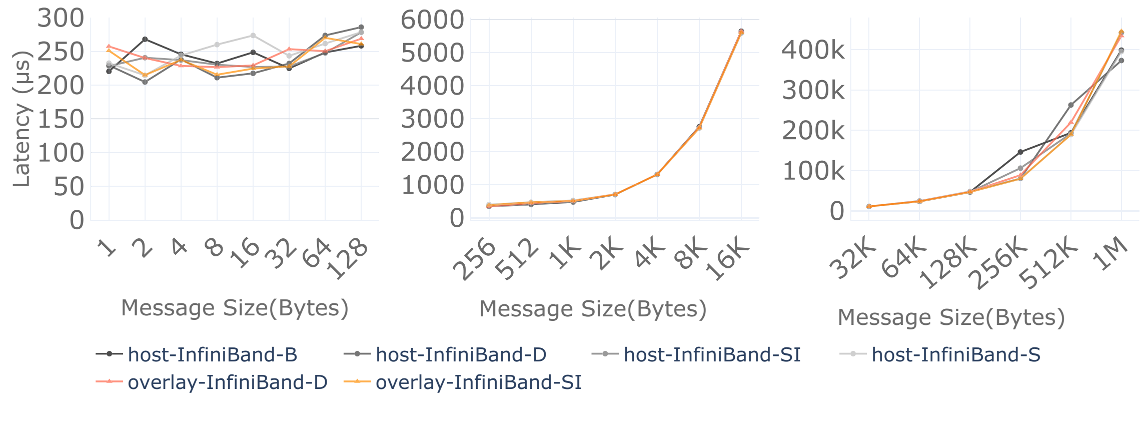
Fig. 3 TCP/IP over Infiniband: Latency (**a**) and bandwidth (**b**) for scenario SCE1 with different networking modes

instance with several network fabrics and protocols, namely TCP/IP, IPoIB, and RDMA.

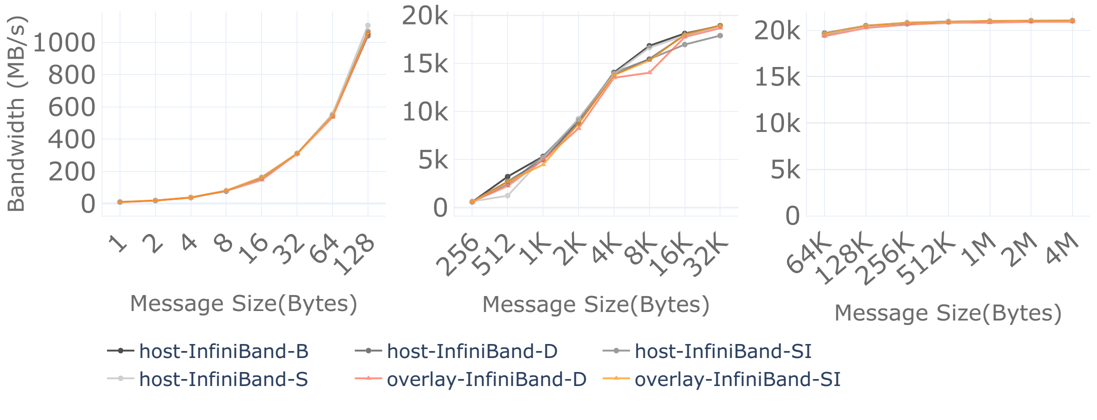
For small and medium messages, we observe that scenario SCE1 has the lowest latency and running more containers per host increases the latency. This increment is related with the number of containers per host only for TCP/IP and IPoIB. In particular, the latency on Docker with message size 8B in scenarios SCE2–SCE6 over TCP/IP, IPoIB, and RDMA has increased by 2%–5%–6%–7%–9%, 8%–13%–16%–17%–18%, 64%–13%–42%–24%–52% compared to SCE1, respectively.

For large messages, TCP/IP has similar performance with different container granularity. However, for IPoIB and RDMA, scenarios with several containers per host (SCE2–SCE6) show up to 19% and 10% lower latency than SCE1 for IPoIB and RDMA, respectively. This is because the memory latency becomes a critical factor when the network latency is not the dominant bottleneck, as occurs in high-speed networks. As shown in Fig. 8, which depicts relevant performance counters of osu-alltoallv for IPoIB and RDMA with large message size (1 MB) on Docker, scenarios SCE2–SCE6 show better cache utilization, fewer local memory accesses, and fewer remote memory accesses than SCE1. These are consequences of the scheduling of the containers (i.e. the cgroups) and their corresponding MPI processes. With scenarios SCE2–SCE6 running more containers, each of them runs fewer processes, tending to a single-level scheduling (i.e. at the cgroup level), which is simpler and allows exploiting processor affinity better, thus improving the cache usage and enforcing local memory accesses.

The memory contention also affects the performance on IPoIB and RDMA. In order to measure the memory contention that occurs on osu-alltoallv with large message size, we calculate the memory contention ratio among cores by using the model proposed by Tudor and Teo [23]. Like those authors, we are not interested in the absolute value of stall cycles, but on how stall cycles grow relative to a baseline value on one core (where there is no contention) due to memory contention among cores. Consequently, we derive the memory contention ratio ω as the stall cycles due to contention divided by the useful work cycles



(a) OSU MPI_Alltoallv Latency



(b) OSU Bidirectional Bandwidth

Fig. 4 **RDMA over Infiniband:** Latency (**a**) and bandwidth (**b**) for scenario SCE1 with different networking modes

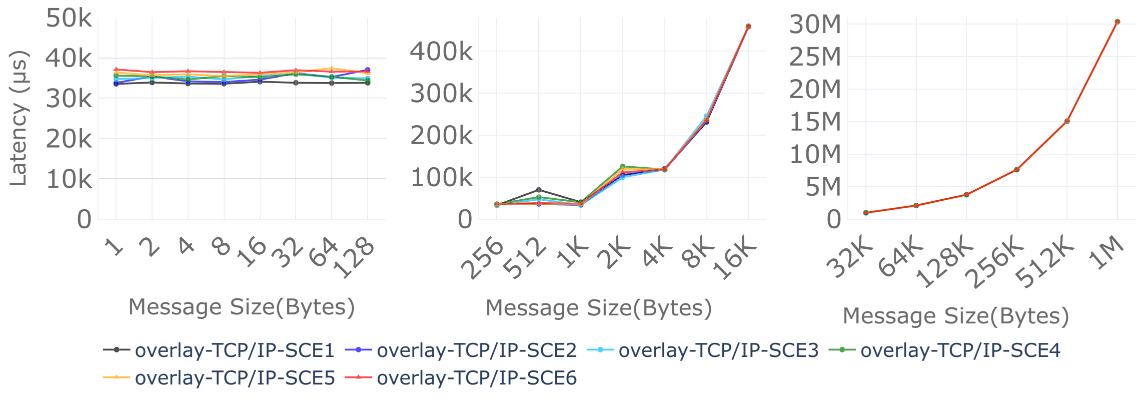
(including stall cycles that are not due to resource contention). Figure 9 presents the average memory contention ratio of osu-alltoallv for IPoIB and RDMA with message size 1 MB on Docker, where a higher ω means more memory contention. As shown in the figure, the memory contention ratio decreases when increasing the number of containers. This is because, as described previously, using more containers decreases the number of accesses to the l3 cache and the memory, which reduces the contention.

4.3.2 HPCC MPI communication-intensive workloads

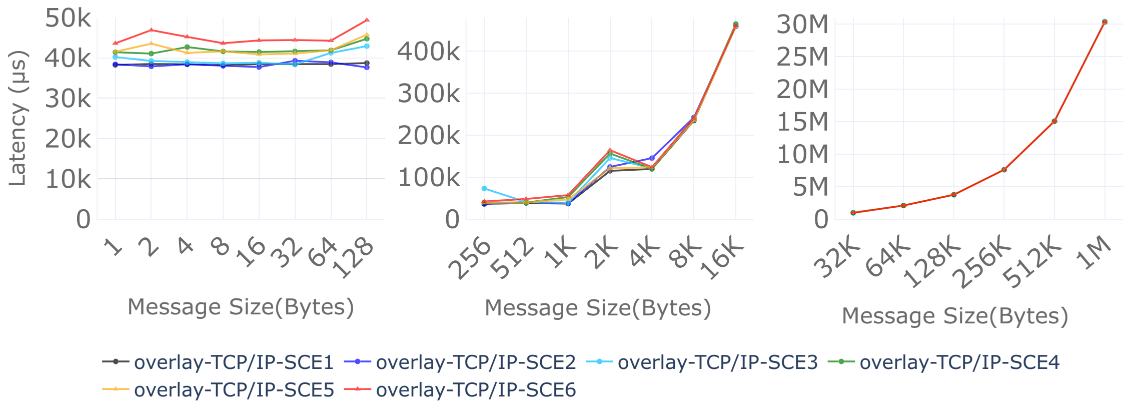
RandomRing Bandwidth benchmark features a number of communication patterns (e.g., non-blocking and blocking concurrent transfers). In particular, it performs MPI_Isend and MPI_Irecv to left and right partner, as well as MPI_Sendrecv, and saves the minimum of both latencies for all rings. We show the results for different container granularity in Figs. 10a, 11a, and 12a. Containerization technologies using overlay networks present significant degradation for TCP/IP and, especially, IPoIB regarding bare-metal and Singularity. As discussed in the previous section, this is due to the overhead introduced by the encapsulation of network packets.

For TCP/IP and IPoIB, increasing the number of containers per host does not have a noticeable impact on the bandwidth. This is because the bottleneck for TCP/IP and IPoIB comes from the interconnection between nodes, which is far slower than the interconnection between containers in the same node or between processes in the same container (i.e., shared-memory). This can be confirmed in Fig. 13a and b. However, for RDMA, multi-container scenarios SCE2–SCE6 have 17%–23%–25%–26%–27% performance degradation in the bandwidth regarding SCE1. This occurs because the interconnection between nodes on RDMA is as fast as the shared-memory communication within a container as shown in Fig. 13c. Therefore, the interconnection between containers in the same node becomes the performance bottleneck, and this increases with the number of containers per node.

G-PTRANS and G-RandomAccess present different point-to-point communication patterns and use different



(a) MPI_Alltoallv latency on Docker

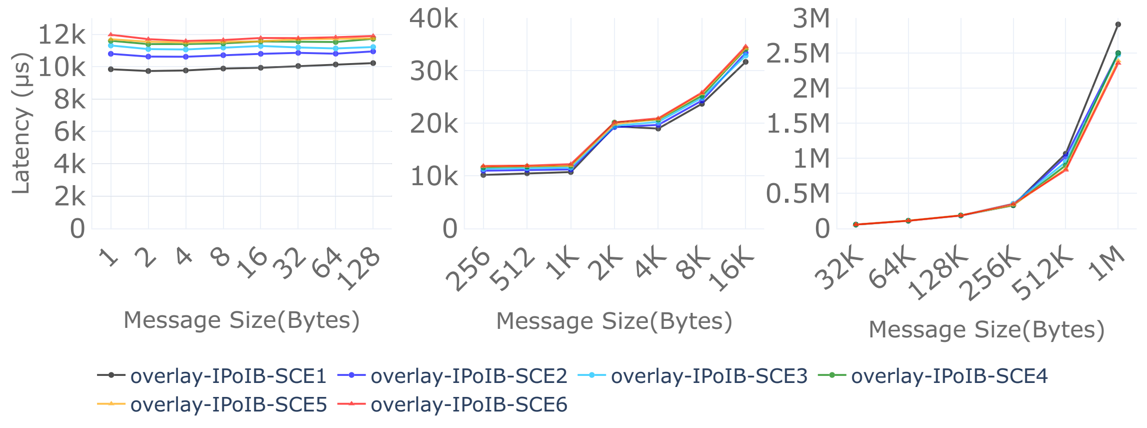


(b) MPI_Alltoallv latency on Singularity-instance

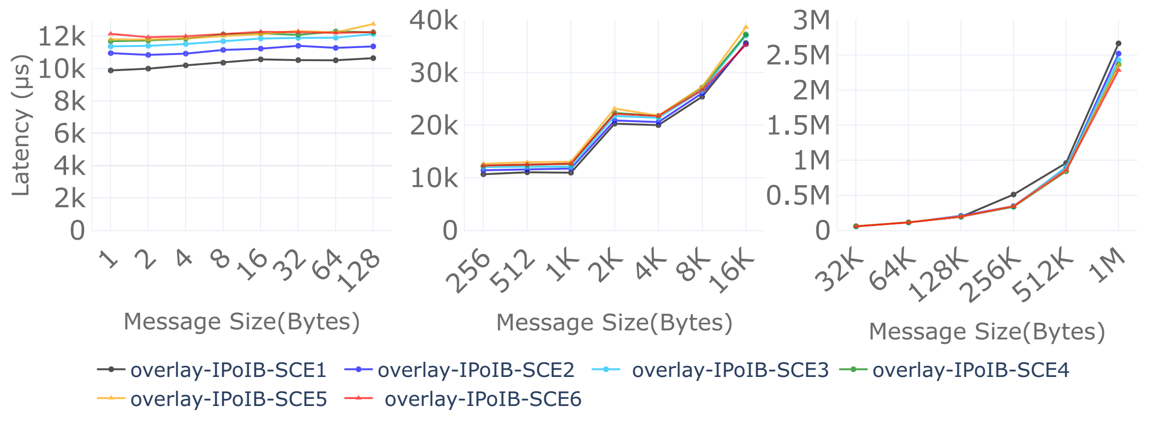
Fig. 5 **TCP/IP over Ethernet:** MPI_Alltoallv latency for multi-container deployment scenarios (SCE1–SCE6)

message sizes. In particular, G-PTRANS performs mainly blocking concurrent transfers (e.g., MPI_Sendrecv) with message size 2 MB. G-RandomAccess uses mostly small-sized non-blocking communication (e.g., MPI_Isend, MPI_Irecv, MPI_Wait). Thus, G-PTRANS is mainly a network-bandwidth-intensive benchmark that behaves similar to RandomRing. In particular, as shown in Figs. 10b, 11b, and 12b, Docker multi-container scenarios SCE2–SCE6 incur 7%–14%–14%–16%–17% performance degradation on RDMA compared to SCE1. On the other side, G-RandomAccess accesses data from all the processes. As shown in Figs. 10c, 11c, and 12c, there is an increasing performance degradation with finer-grained containers. In particular, Docker multi-container scenarios SCE2–SCE6 have 6%–11%–13%–15%–14%, 12%–16%–21%–22%–21%, and 8%–23%–25%–34%–38% performance degradation regarding SCE1, for TCP/IP, IPoIB, and RDMA, respectively. This occurs because G-RandomAccess performs a high number of MPI invocations, and when increasing the number of containers a significant part of them involve inter-container communications instead of intra-container (which are faster). This is especially relevant for RDMA, given that the memory latency is a critical parameter for the performance of G-RandomAccess.

G-FFT mainly uses MPI_Alltoall communication pattern to transfer large data, and it is also intensive on memory bandwidth and computation. As shown in Figs. 10d, 11d, and 12d, the performance of multi-container scenarios SCE2–SCE6 is similar on TCP/IP and IPoIB, whereas on RDMA they show some performance degradation compared to SCE1 (e.g., around 8% in average on Docker). Whereas the performance on TCP/IP (and IPoIB) is mostly limited by the network bandwidth, the performance on RDMA depends on the memory latency, which is worse when running multiple containers per host. However, this incurs low degradation due to the low number of MPI invocations performed by G-FFT.



(a) MPI_Alltoallv latency on Docker



(b) MPI_Alltoallv latency on Singularity-instance

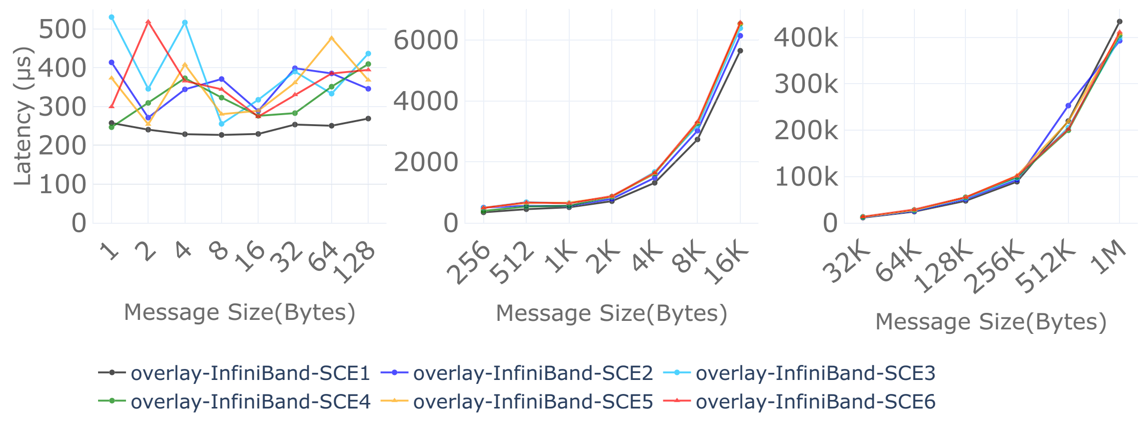
Fig. 6 **TCP/IP over Infiniband:** MPI_Alltoallv latency for multi-container deployment scenarios (SCE1–SCE6)

4.3.3 HPCC MPI throughput workloads

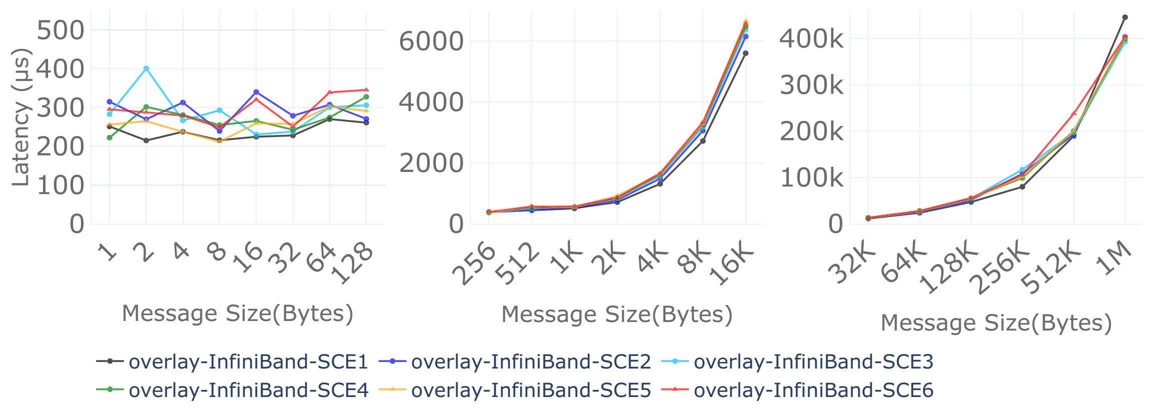
EP-STREAM characterizes the memory bandwidth, while EP-DGEMM stresses the computation capabilities of the system. As shown in Figs. 14, 15 and 16, overlay networking does not bring explicit performance penalties due to the low amount of interprocess communication. Similarly, multi-container scenarios do not show significant performance differences, except EP-DGEMM on SCE6. In this scenario, EP-DGEMM on Docker (also on Singularity-instance + cgroup) shows noticeable performance improvement (11%, 16%, and 7% for TCP/IP, IPoIB, and RDMA, respectively) regarding other deployment scenarios (including bare-metal). This is a consequence of the scheduling of the containers (i.e., cgroups) and their corresponding MPI processes. As each container runs a single process, this is essentially a single-level scheduling (i.e. at the cgroup level), which is simpler and allows to exploit processor affinity better, in a similar way to when processes are pinned explicitly.

4.4 Impact of affinity on multi-container per host deployment scenarios with different network fabrics

Figures 17, 18, and 19 show the performance results of Docker multi-container deployment scenarios with different affinity settings for various network interconnects and protocols. Singularity-instance shows similar results, which have not been included due to space constraints. As discussed in Sect. 4.3, communication-intensive benchmarks have degradation in multi-container deployment scenarios due to the overhead of overlay communication for TCP/IP and IPoIB. Similarly, RDMA is limited by the bandwidth of inter-container communication. Setting affinity cannot avoid this performance degradation. For example, enabling affinity on G-PTRANS does not bring improvements



(a) MPI_Alltoallv latency on Docker



(b) MPI_Alltoallv latency on Singularity-instance

Fig. 7 **RDMA over Infiniband:** MPI_Alltoallv latency for multi-container deployment scenarios (SCE1–SCE6)

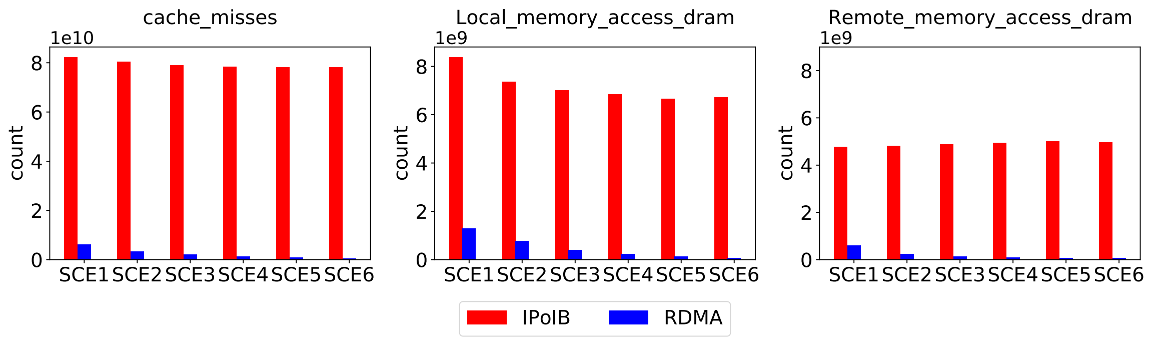


Fig. 8 Performance event counters of osu-alltoallv for different interconnects and protocols with message size 1 MB on Docker

because its performance is mainly limited by the network bandwidth.

The effectiveness of affinity in multi-container deployments depends significantly on the resource usage characteristics of each benchmark. For example, restricting the range of CPUs to be assigned to the containers can help applications that suffer many cpu-migrations and context-switches. Restricting the memory access of the containers to the NUMA node where their CPUs belong can help applications presenting an elevated number of remote memory accesses.

CPU and memory affinity have considerably increased the performance of EP-DGEMM in all the scenarios. Specifically, the speedup in CPU, CPUMEM, and

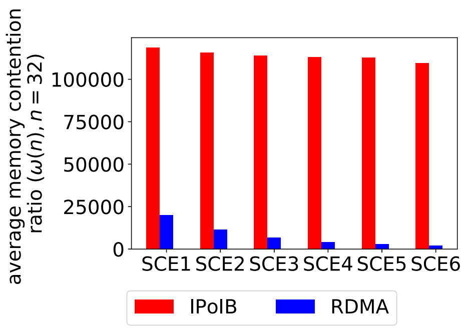


Fig. 9 Average memory contention ratio of osu-alltoallv for different interconnects and protocols with message size 1 MB on Docker

CPUMEMPIN scenarios with respect to ANY scenarios ranges from 9%–21% (SCE2–SCE5 CPU), 18%–35% (SCE2–SCE5 CPUMEM), and 22%–41% (SCE1–SCE6 CPUMEMPIN) on TCP/IP; 15%–26% (SCE2–SCE5 CPU), 19%–36% (SCE2–SCE5 CPUMEM), and 21%–42% (SCE1–SCE6 CPUMEMPIN) on IPoIB; and 17%–22% (SCE2–SCE5 CPU), 21%–37% (SCE2–SCE5 CPU-MEM), and 31%–43% (SCE1–SCE6 CPUMEMPIN) on RDMA. These performance increments are directly related with the container granularity, as finer-grained deployments provide better speedup on CPU and CPUMEM. This happens because CPU affinity restricts the number of assigned CPUs within each container, hence the processes running in finer-grained containers have less available CPUs where they could be migrated. Setting CPU affinity reduces the number of context-switches and cpu-migrations in CPUX scenarios, while setting memory affinity restricts as well the remote memory accesses in CPUMEMX scenarios.

Benchmarks with different memory usage characteristics can benefit from setting affinity. This is more explicit in RDMA scenarios, where the computation and memory latency also become critical parameters instead of only the network interconnect. In particular, G-FFT on RDMA has significant performance improvement, 18%–32% (SCE2–SCE5 CPUMEM) and 16%–32% (SCE1–SCE6 CPU-MEMPIN). As the all-to-all communication on RDMA is considerably faster than on TCP/IP, the overall performance is impacted then by the memory latency. Therefore, G-FFT on RDMA benefits from multi-container deployments with memory affinity which enforces local memory accesses.

Noticeably, setting affinity decreases the performance of G-RandomAccess on TCP/IP, up to 21% (CPU), 22% (CPUMEM), and 24% (CPUMEMPIN). By analyzing the results in Figs. 20 and 21, we found out that the actual cause of the performance degradation of CPUMEMPIN was the load imbalance among processes. Figure 20, which depicts the time spent in MPI communication patterns of

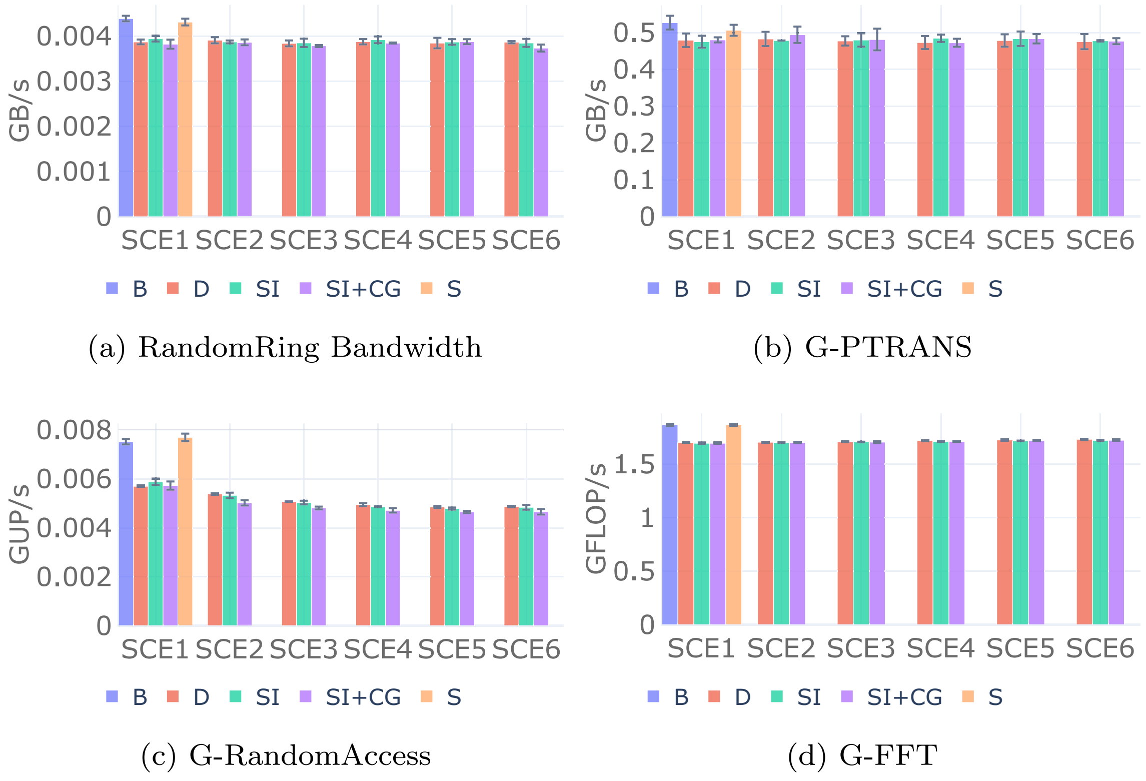


Fig. 10 **TCP/IP over Ethernet:** Impact of container granularity in HPCC MPI communication workloads

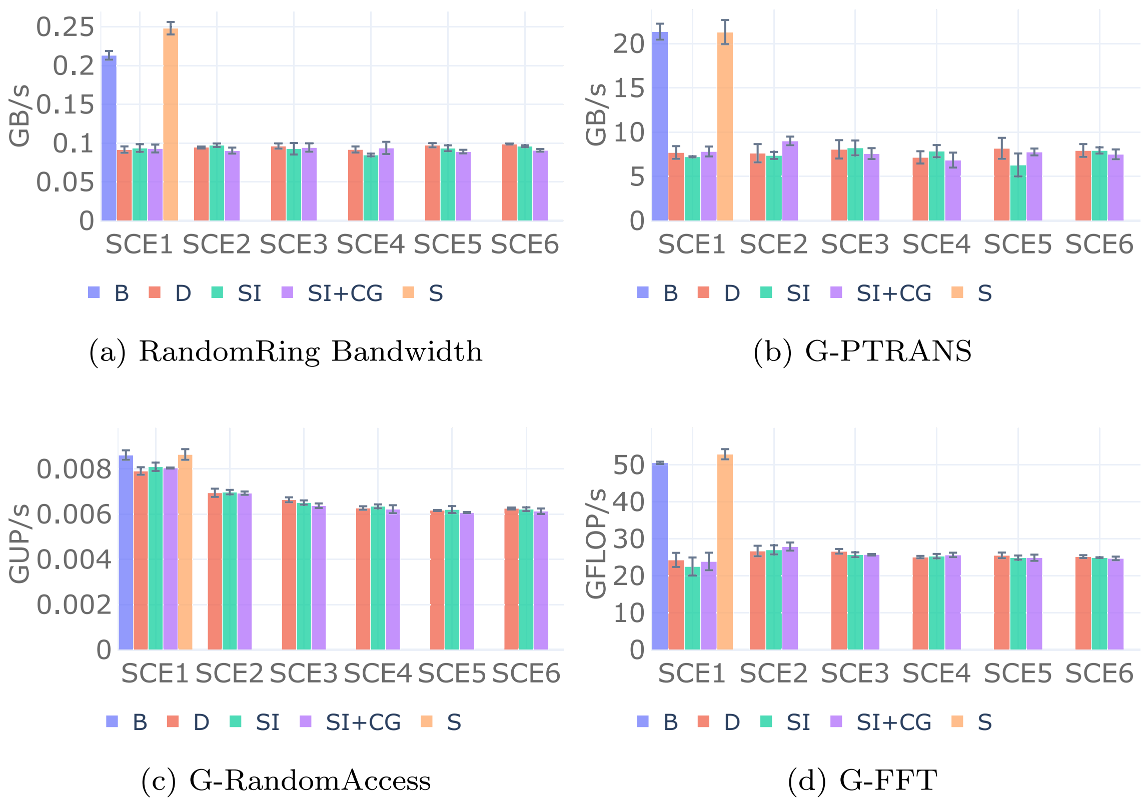


Fig. 11 **TCP/IP over InfiniBand:** Impact of container granularity in HPCC MPI communication workloads

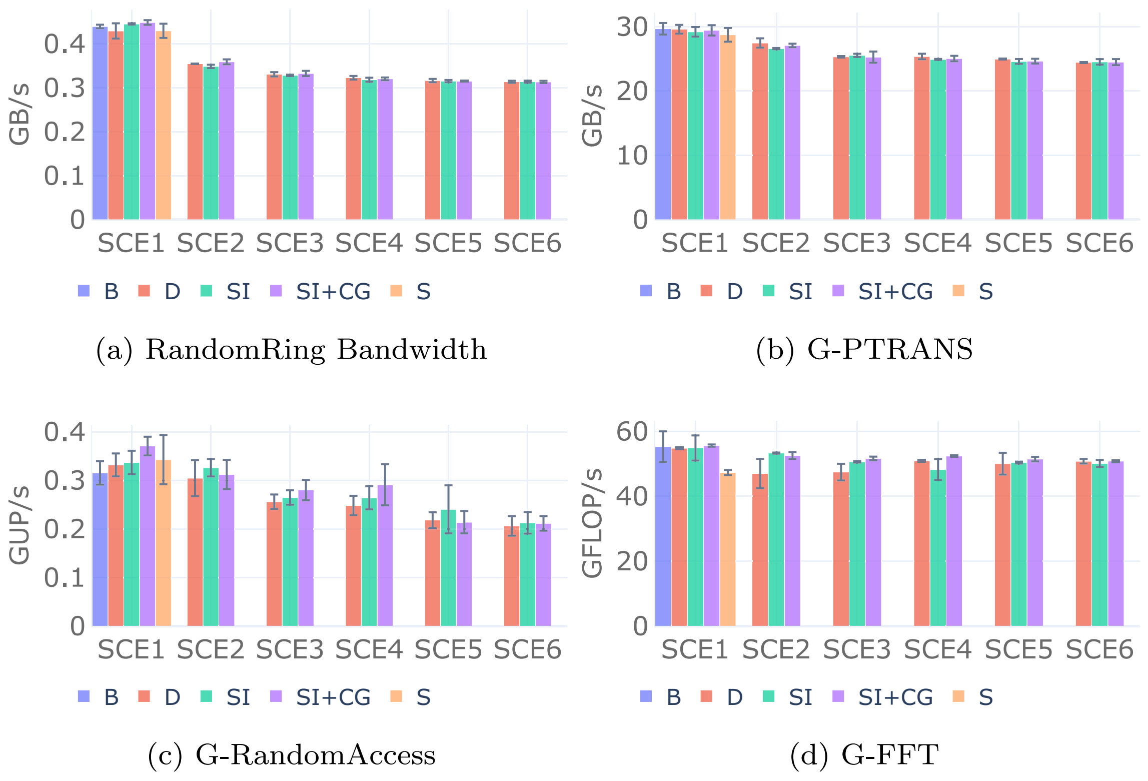
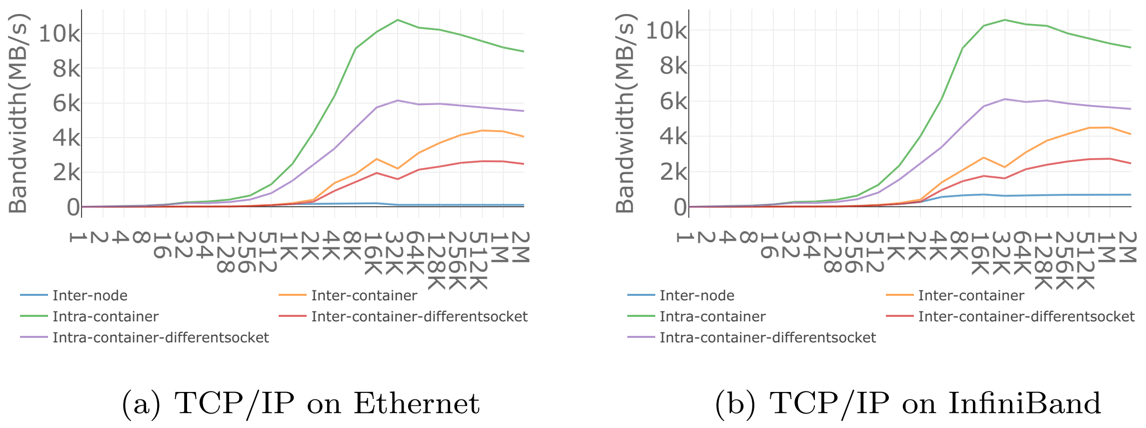


Fig. 12 **RDMA over InfiniBand:** Impact of container granularity in HPCC MPI communication workloads



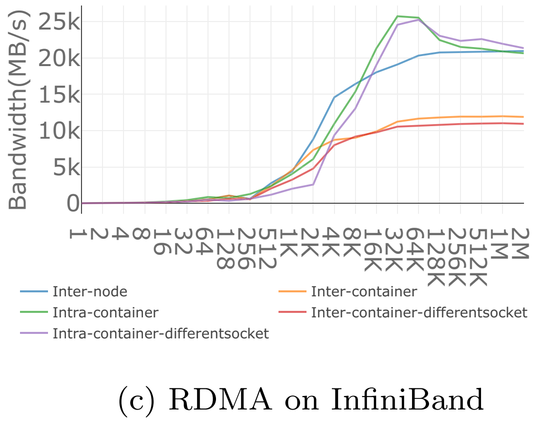


Fig. 13 Maximum aggregate bandwidth of inter-node, inter-container, and intra-container communications with different network protocols

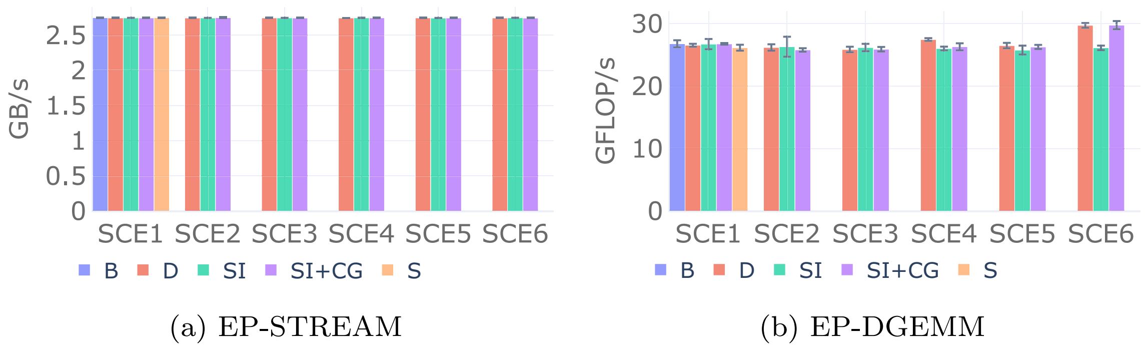
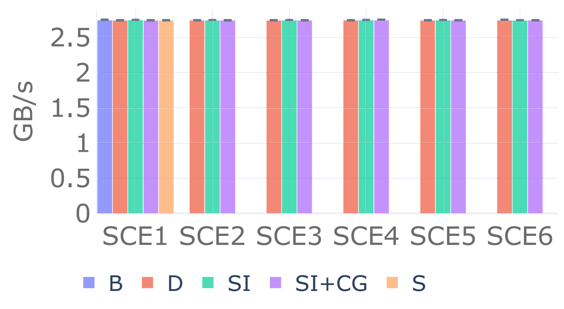


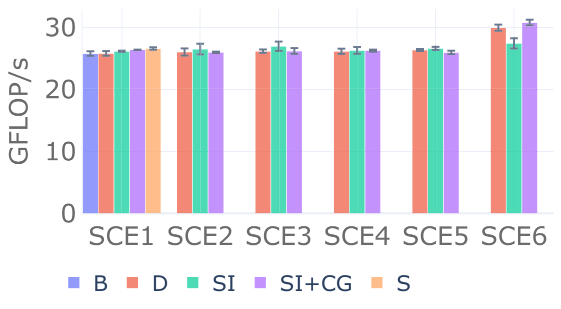
Fig. 14 **TCP/IP over Ethernet:** Impact of container granularity in HPCC MPI throughput workloads

G-RandomAccess for TCP/IP interconnect with ANY and CPUMEMPIN affinities on Docker scenario SCE1, shows that the time spent on MPI_Waitany and especially MPI_Barrier for CPUMEMPIN is much higher than ANY. Thus, this requires us to generate the MPI profile by duration time of the experiments.

Figure 21, which shows a detailed MPI duration profile for all 128 processes, reveals that in ANY, the scheduler can better balance the load among processes and reduce their wait time in the barrier. Contrariwise, in CPUMEMPIN, some processes incur high wait latency that slows down the entire application. Given the random nature of the data accesses in G-RandomAccess benchmark, some processes might receive more requests than others, but as they are pinned to specific cores they cannot take advantage of other cores which are currently idle, thus causing the busy-waiting of other processes and introducing more latency. ANY affinity can mitigate this problem by

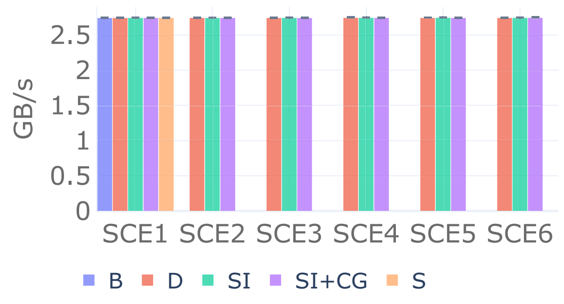


(a) EP-STREAM

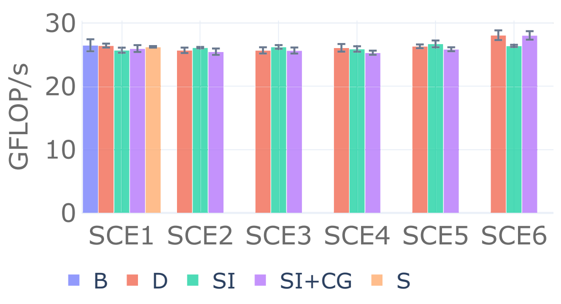


(b) EP-DGEMM

Fig. 15 **TCP/IP over InfiniBand:** Impact of container granularity in HPCC MPI throughput workloads

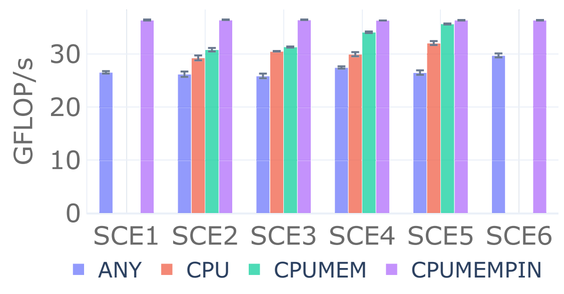


(a) EP-STREAM

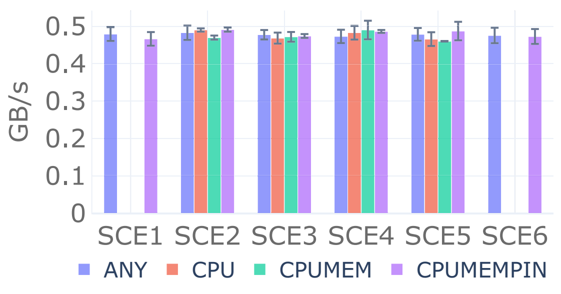


(b) EP-DGEMM

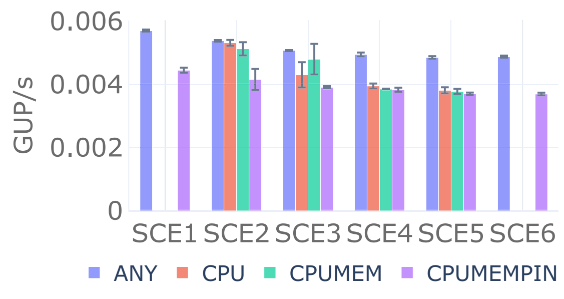
Fig. 16 **RDMA over InfiniBand:** Impact of container granularity in HPCC throughput workloads



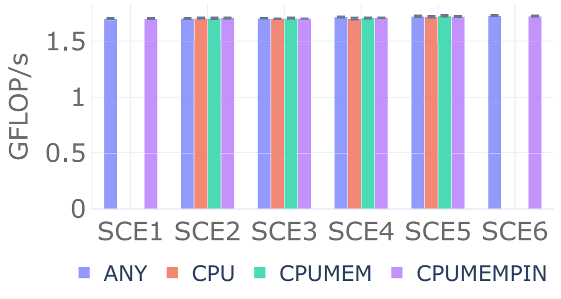
(a) EP-DGEMM



(b) G-PTRANS

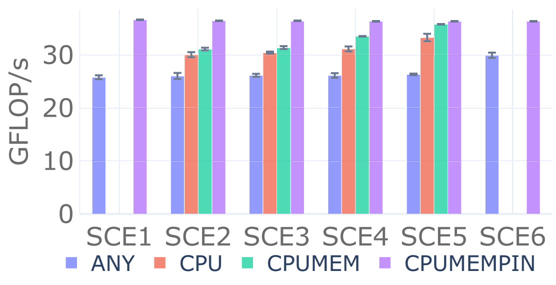


(c) G-RandomAccess

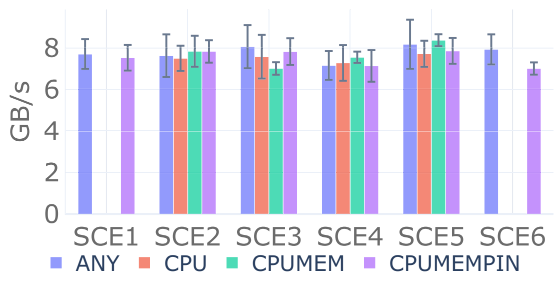


(d) G-FFT

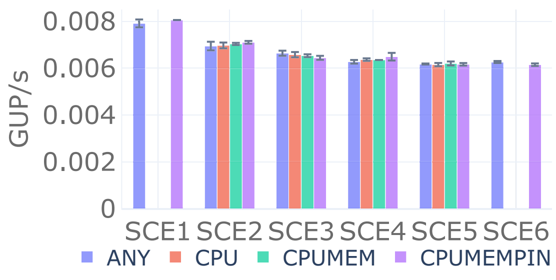
Fig. 17 **TCP/IP over Ethernet:** Impact of affinity for multi-container deployments of HPCC MPI workloads



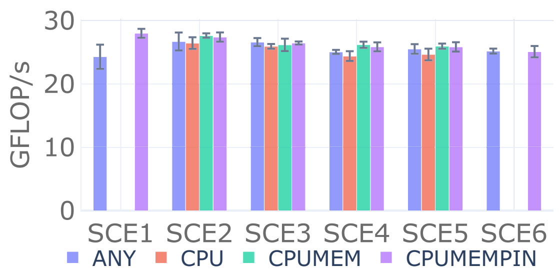
(a) EP-DGEMM



(b) G-PTRANS

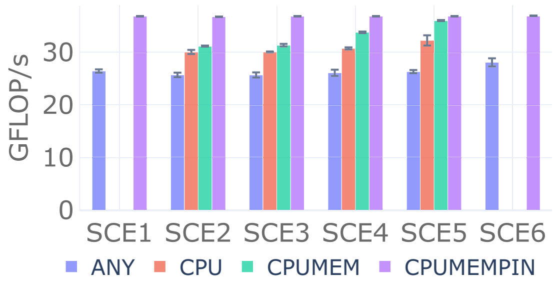


(c) G-RandomAccess

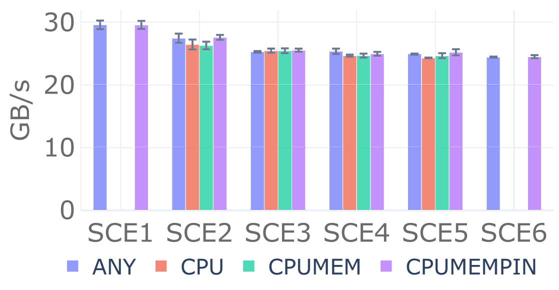


(d) G-FFT

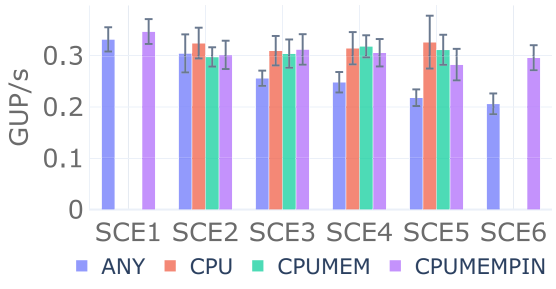
Fig. 18 **TCP/IP over InfiniBand:** Impact of affinity for multi-container deployments of HPCC MPI workloads



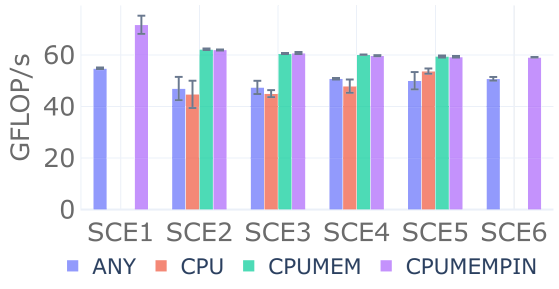
(a) EP-DGEMM



(b) G-PTRANS



(c) G-RandomAccess



(d) G-FFT

Fig. 19 **RDMA over InfiniBand:** Impact of affinity for multi-container deployments of HPCC MPI workloads

allowing to migrate processes to achieve better load balance. However, enabling affinity can help to improve the performance on RDMA, in particular, 7%–50% (SCE2–SCE5 CPU), 0–43% (SCE2–SCE5 CPUMEM), and 0–43% (SCE1–SCE6 CPUMEMPIN). With RDMA, memory latency becomes relevant for performance, and for this reason, restricting the remote memory accesses through affinity can reduce the degradation.

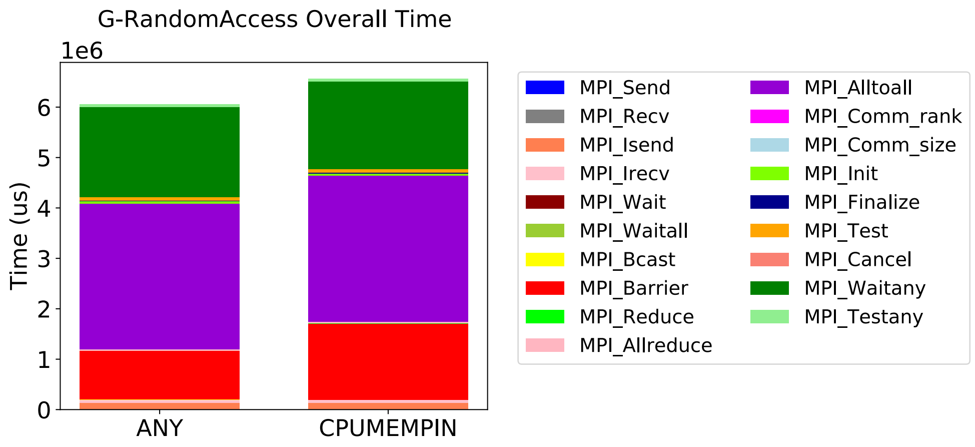


Fig. 20 Time spent in MPI communication patterns of G-RandomAccess for TCP/IP interconnect with ANY and CPUMEMPIN affinity on Docker-SCE1

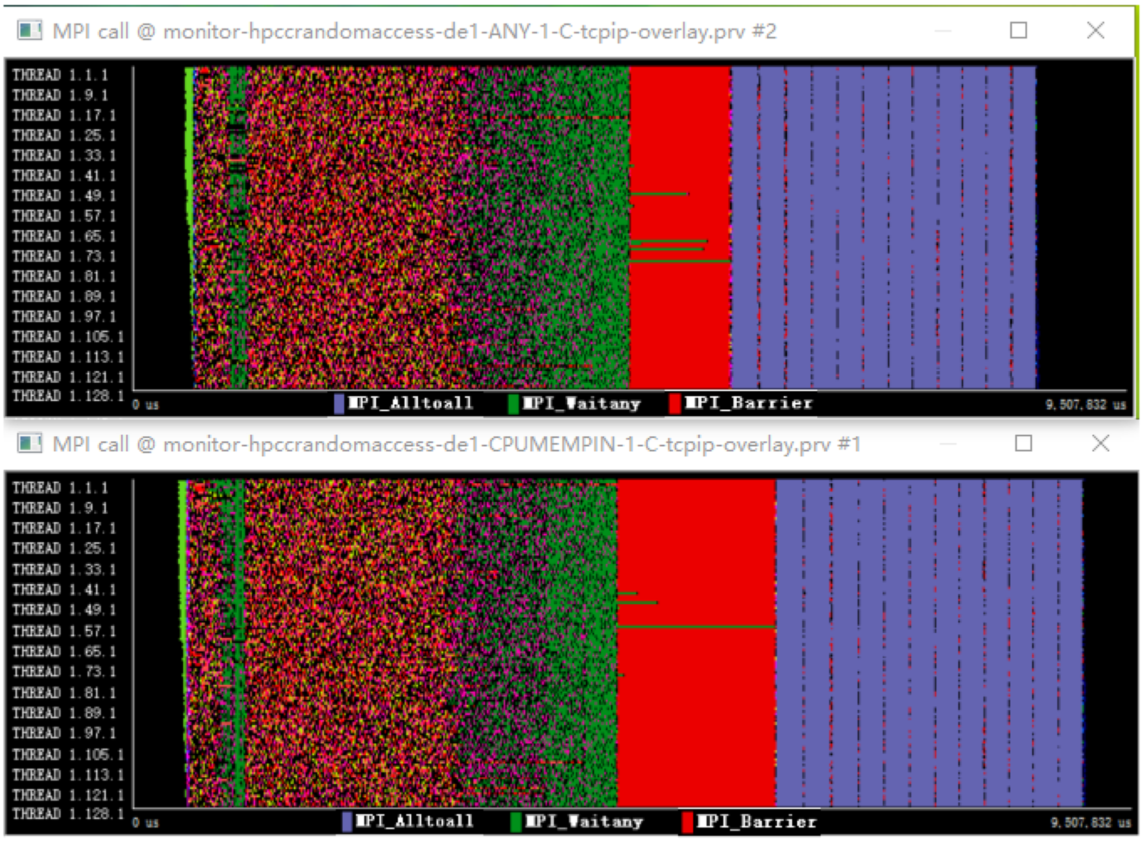


Fig. 21 MPI profile of G-RandomAccess for TCP/IP interconnect with ANY (top) and CPUMEMPIN (bottom) affinity on Docker-SCE1

4.5 Performance insights on multi-container per host deployment scenarios with different network fabrics in large-scale clusters

Experiments in previous sections were run in a testbed with five nodes (1 master + 4 workers). Nevertheless, we anticipate that most of the performance insights obtained in those sections would still hold for multi-container deployment scenarios with different network fabrics in a large-scale cluster.

First, we expect default Singularity to have close to bare-metal performance because it can use an underlay networking approach. However, it cannot support multi-container deployments.

Second, we expect Docker and Singularity-instance, which can support multi-container deployments by means of an overlay networking approach, to incur noticeable performance degradation for MPI communication workloads. This degradation is expected to increase as a function of the number of nodes, as this will increase the proportion of inter-node communications. The latter can be appreciated in Fig. 22, which shows the distribution of invocations to function alltoallv in the OSU alltoallv benchmark among inter-container, intra-container, and inter-node communications with different number of nodes and deployment scenarios. Note that this benchmark communicates all the processes so the impact might vary depending on the communication pattern for other applications. It will basically depend on their ratio among inter-container, intra-container, and inter-node communications. Furthermore, the degradation will be more noticeable for TCP on Ethernet and on Infiniband, as they provide much worse performance than RDMA (check inter-node bandwidth in Fig. 13) and the performance difference grows rapidly as the number of nodes increases [24].

As also shown in Fig. 22, fine-grain multi-container deployments will transform intra-node communications using shared-memory on inter-container communications. Although Fig. 13 showed that inter-container communications are slower, this effect will be diluted in large-scale clusters given the dominance of inter-node communications, hence multi-container deployments should show similar behavior in terms of the performance of deployment schemes with different container granularity.

Third, although setting affinity cannot avoid the overhead incurred by overlay networking, we expect that it can also make a difference on MPI throughput workloads in large-scale clusters, as the performance bottlenecks for those applications are the computation and memory allocation and not the network transfers. As shown in Fig. 23, which displays the performance of EP-DGEMM using multi-container deployments scenarios when running on a testbed with 7 nodes (1 master + 6 workers), affinity still brings valuable performance benefits in all multi-container deployment scenarios and network fabrics.

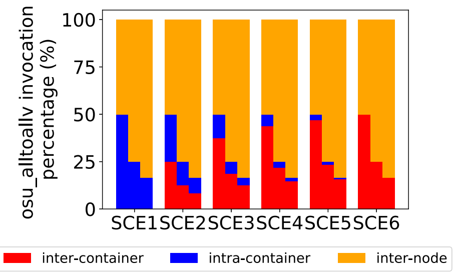


Fig. 22 Distribution of invocations to alltoallv among inter-container, intra-container, and inter-node communications on a different number of nodes (each group shows the percentage for 2, 4, and 6 worker nodes, respectively)

5 Related work

Recent works have evaluated Docker and Singularity as candidate containerization technologies to run HPC applications [6, 25]. These works mainly focused on a single container wrapped HPC application allocated on a single host, but without considering different container granularity and different container interconnects. Rudyy et al. [26] discussed the execution of a given containerized HPC application on HPC clusters, and mainly studied different container technologies and different HPC architectures, but did not consider different container granularity.

Zhang et al. [4, 8] studied the performance characterization of KVM and Docker for running HPC applications on SR-IOV enabled InfiniBand clusters, and in a further work [9], they stated that Singularity-based container technology is ready for running MPI applications on HPC clouds. Also in their work [13], they studied the locality and NUMA aware MPI runtime for nested virtualization (a combination of virtual machines and containers). These works evaluated different aspects of using containerization for HPC applications, but none of them considered deployment schemes with different container granularity.

Chung et al. [11] evaluated Docker containers for deploying MPI applications. They proposed deployment scenarios with different container granularity. However, this work only tested computing intensive and data intensive applications and did not consider InfiniBand networks. Their further work [12] considered Docker on InfiniBand and highlighted the benefits of using InfiniBand with Docker. This work showed the results of several benchmarks, but did not consider affinity or different network fabrics and protocols.

Saha et al. [7] evaluated the performance of running HPC applications using Docker Swarm. Whereas they considered a different number of MPI ranks distributed in multiple containers across multiple hosts, their latency experiments only include a fixed message size (e.g., 65536 bytes). Their results showed that deploying one rank per container had worse performance because they ignored the binding policy.

Saha et al. [10] enabled the orchestration of MPI applications with Apache Mesos, and provided a policy-based approach for deploying MPI ranks on containers with different granularity. However, this policy is based on TCP/IP over Ethernet, and does not consider InfiniBand. Beltre et al. [2] evaluated Kubernetes to run MPI applications in clouds. They compared TCP/IP and InfiniBand, but they did not include multi-container deployments.

In our previous work [16], we enabled the interconnection across hosts through TCP/IP protocol between Singularity instances running Big Data applications.

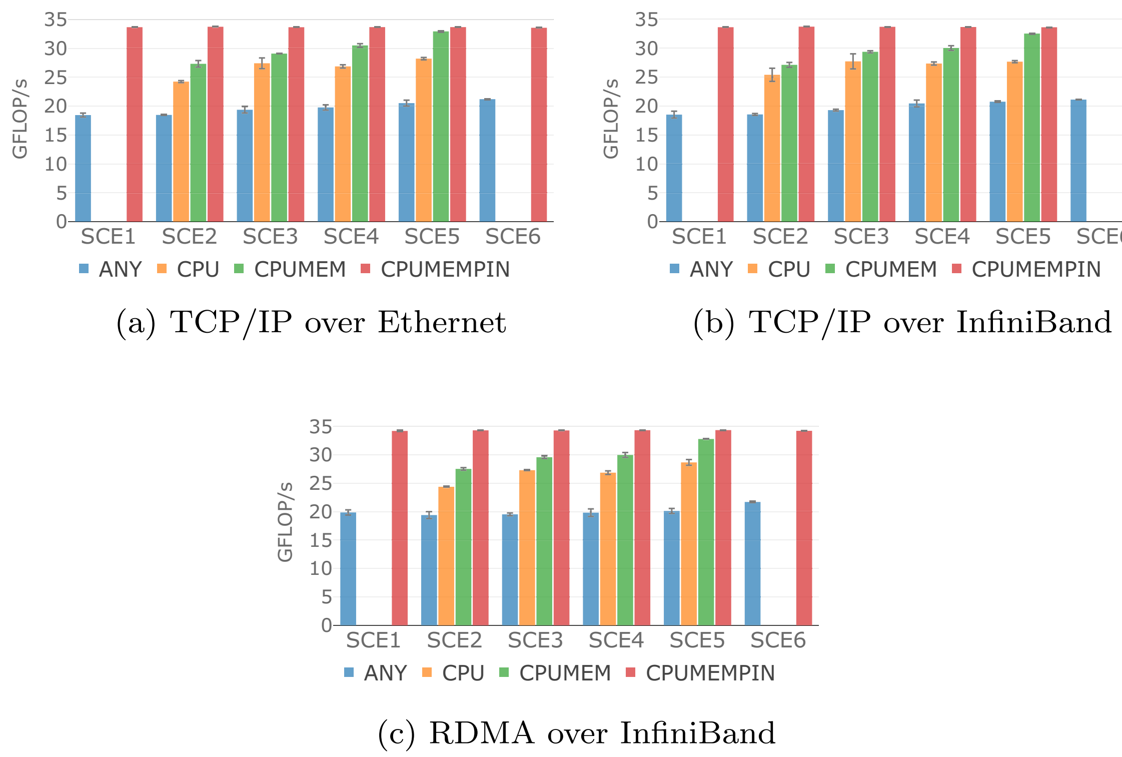


Fig. 23 Performance of EP-DGEMM using multi-container deployments scenarios with different network fabrics when running on a testbed with 7 nodes (1 master + 6 workers)

Moreover, in our latest work [3], we performed a performance analysis of multi-container deployments with different container granularity for a number of HPC applications, but only with a single host and the TCP/IP protocol.

Existing literature shows approaches and results of deploying a single container per host using Docker or Singularity in the cloud, and most of the work considers using the orchestration thus ignoring the original impact of the network fabric and protocols. Moreover, there still exists a gap in terms of multi-container per host deployments evaluation on an InfiniBand cluster which considers the performance of different container granularity and enhanced affinity settings using different network fabrics and protocols for HPC workloads.

6 Conclusion and future work

This paper has presented a performance characterization of different containerization technologies (including Docker and Singularity) for HPC workloads on InfiniBand clusters from four dimensions, namely network interconnects (including Ethernet and InfiniBand) and protocols (including TCP/IP and RDMA), networking modes (including host, MACVLAN, and overlay networking), and processor and memory affinity. We focus especially on understanding how the container granularity and its combination with processor and memory affinity impact the performance when using different networking modes. We used OSU benchmarks to measure the network performance considering different message sizes, as well as HPCC workloads that exhibit different communication patterns, memory accesses, and computation.

We concluded that default Singularity has close to bare-metal performance because it can use an underlay networking approach. However, it does not support fine-grain multi-container deployments, which are only possible with Docker and Singularity-instance. These use an overlay networking approach, which incurs noticeable performance degradation for MPI communication workloads, and show similar behavior in terms of the performance of deployment schemes with different container granularity and affinity. In particular, fine-grain multi-container deployments transform intra-node communications using shared-memory on inter-container communications, which could increase the network latency of some MPI operations, but can alleviate the latency and contention of memory accesses when these are the performance bottleneck.

Setting affinity cannot avoid the overhead incurred by overlay networking, but it can make a difference on MPI throughput workloads and even MPI communication workloads where the computation and memory allocation have replaced the network transfers as the performance bottlenecks (e.g. when running on RDMA). In those scenarios, we have shown how processor and memory affinity can reduce the number of kernel-level cycles spent due to the process preemption (i.e., avoid cpu-migrations and context-switches) and due to the system calls (i.e., exploit locality in data accessing).

In the future, these insights about the performance of multi-container deployments on InfiniBand clusters, especially those regarding the impact of the container granularity and affinity with different networking modes, can be employed to derive placement policies when deploying HPC workloads which can get better utilization of the resources while maintaining application performance. Those policies could be integrated in traditional HPC job schedulers, such as Slurm,[10] which have also already started to support containers, as well as, new schedulers for HPC workloads with native containerization support, such as the Kubernetes native batch scheduling system (i.e., Volcano[11]). Both approaches would allow integrating our deployment schemes, namely fine-grained container granularity, affinity, and overlay networking, with the traditional HPC scheduling capabilities and QoS requirements supported by those schedulers.

Acknowledgements We thank Lenovo for providing the testbed to run the experiments in this paper. This work was partially supported by Lenovo as part of Lenovo-BSC collaboration agreement, by the Spanish Government under contract PID2019-107255GB-C22, and by the Generalitat de Catalunya under contract 2017-SGR-1414 and under Grant No. 2020 FI-B 00257.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Iosup, A., Ostermann, S., Yigitbasi, M.N., Prodan, R., Fahringer, T., Epema, D.: Performance analysis of cloud computing services for many-tasks scientific computing. IEEE Trans. Parallel Distrib. Syst. **22**(6), 931–945 (2011). https://doi.org/10.1109/TPDS.2011.66
2. Beltre, A.M., Saha, P., Govindaraju, M., Younge, A., Grant, R.E.: Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms. In: Proceedings of CANOPIE-HPC 2019: 1st International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC, pp. 11–20 (2019). https://doi.org/10.1109/CANOPIE-HPC49598.2019.00007
3. Liu, P., Guitart, J.: Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study. Journal of Supercomputing (2020). https://doi.org/10.1007/s11227-020-03518-1
4. Zhang, J., Lu, X., Panda, D.K.: High performance MPI library for container-based HPC cloud on InfiniBand clusters. In: 45th International Conference on Parallel Processing (ICPP), pp. 268–277. IEEE (2016). https://doi.org/10.1109/ICPP.2016.38
5. Ibrahim, K.Z., Hofmeyr, S., Iancu, C.: The case for partitioning virtual machines on multicore architectures. IEEE Trans. Parallel Distrib. Syst. **25**(10), 2683–2696 (2014). https://doi.org/10.1109/TPDS.2013.242
6. Arango, C., Dernat, R., Sanabria, J.: Performance evaluation of container-based virtualization for high performance computing environments. CoRR (2017). arXiv:1709.10140
7. Saha, P., Beltre, A., Uminski, P., Govindaraju, M.: Evaluation of Docker Containers for Scientific Workloads in the Cloud. In: Proceedings of Practice and Experience on Advanced Research Computing (PEARC18). ACM, New York (2018). https://doi.org/10.1145/3219104.3229280
8. Zhang, J., Lu, X., Panda, D.K.: Performance characterization of hypervisor-and container-based virtualization for HPC on SR-IOV enabled infiniband clusters. In: Proceedings of 30th International on Parallel and Distributed Processing Symposium (IPDPS16), pp. 1777–1784. IEEE (2016). https://doi.org/10.1109/IPDPSW.2016.178
9. Zhang, J., Lu, X., Panda, D.K.: Is singularity-based container technology ready for running MPI applications on HPC clouds? In: Proceedings of 10th International Conference on Utility and Cloud Computing (UCC17), pp. 151–160. ACM, New York (2017). https://doi.org/10.1145/3147213.3147231
10. Saha, P., Beltre, A., Govindaraju, M.: Scylla: a mesos framework for container based MPI jobs. CoRR (2019). arXiv:1905.08386
11. Chung, M.T., Quang-Hung, N., Nguyen, M.T., Thoai, N.: Using Docker in high performance computing applications. In: Proceedings of 6th International Conference on Communications and Electronics (ICCE16), pp. 52–57. IEEE (2016). https://doi.org/10.1109/CCE.2016.7562612
12. Chung, M.T., Le, A., Quang-Hung, N., Nguyen, D., Thoai, N.: Provision of Docker and InfiniBand in high performance computing. In: Proceedings of the 2016 International Conference on Advanced Computing and Applications, ACOMP16, pp. 127–134. IEEE (2016). https://doi.org/10.1109/ACOMP.2016.027
13. Zhang, J., Lu, X., Panda, D.K.: Designing locality and NUMA aware MPI runtime for nested virtualization based HPC cloud with SR-IOV enabled InfiniBand. SIGPLAN Not. **52**(7), 187–200 (2017). https://doi.org/10.1145/3140607.3050765
14. Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: scientific containers for mobility of compute. PLoS ONE **12**(5), e0177459 (2017). https://doi.org/10.1371/journal.pone.0177459
15. HPC Wire: Sylabs releases singularity 3.0 container platform. Cites AI Support (2018). https://www.hpcwire.com/2018/10/08/sylabs-releases-singularity-3-0-container-platform-cites-ai-support/
16. Sauvanaud, C., Dholakia, A., Guitart, J., Kim, C., Mayes, P.: Big data deployment in containerized infrastructures through the interconnection of network namespaces. Softw. Pract. Exp. **50**(7), 1087–1113 (2020). https://doi.org/10.1002/spe.2793

[10] https://slurm.schedmd.com/containers.html.

[11] https://volcano.sh/en/.

17. Shanley, T.: InfiniBand Network Architecture. Addison Wesley, Boston (2002)
18. Subramoni, H., Lai, P., Luo, M., Panda, D.K.: RDMA over Ethernet: a preliminary study. In: Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER09), pp. 1–9 (2009). https://doi.org/10.1109/CLUSTR.2009.5289144
19. Grun, P.: Introduction to Infiniband for end users. Technical Report. InfiniBand Trade Association (2010)
20. Luszczek, P.R., Bailey, D.H., Dongarra, J.J., Kepner, J., Lucas, R.F., Rabenseifner, R., Takahashi, D.: The HPC Challenge (HPCC) benchmark suite. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC06), pp. 213–es. ACM, New York (2006). https://doi.org/10.1145/1188455.1188677
21. Xing, F., You, H., Lu, C.: HPC benchmark assessment with statistical analysis. Procedia Comput. Sci. **29**, 210–219 (2014). https://doi.org/10.1016/j.procs.2014.05.019
22. HPC Advisory Council: HPCC performance benchmark and profiling (2015). https://hpcadvisorycouncil.com/pdf/HPCC_Analysis_and_Profiling_Intel_E5-2697v3.pdf
23. Tudor, B.M., Teo, Y.M.: A practical approach for performance analysis of shared-memory programs. In: Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium, pp. 652–663 (2011). https://doi.org/10.1109/IPDPS.2011.68
24. HPC Advisory Council: Interconnect analysis: 10GigE and InfiniBand in high performance computing. White Paper (2009). https://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf
25. Younge, A.J., Pedretti, K., Grant, R.E., Brightwell, R.: A tale of two systems: using containers to deploy HPC applications on supercomputers and clouds. In: Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 74–81 (2017). https://doi.org/10.1109/CloudCom.2017.40
26. Rudyy, O., Garcia-Gasulla, M., Mantovani, F., Santiago, A., Sirvent, R., Vázquez, M.: Containers in HPC: a scalability and portability study in production biological simulations. In: Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 567–577 (2019). https://doi.org/10.1109/IPDPS.2019.00066

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Peini Liu received her M.S. degree in College of Computer at National University of Defense Technology (NUDT), in 2018. She is currently a Ph.D. student in Computer Architecture Department of the Universitat Politècnica de Catalunya (UPC) and collaborating with Emerging Technologies for Artificial Intelligence group of Barcelona Supercomputing Center (BSC). Her research interests include virtualization/containerization technologies, cloud native, resource management and the convergence of HPC, Big Data and AI.

Jordi Guitart received the M.S. and Ph.D. degrees in Computer Science at the Universitat Politècnica de Catalunya (UPC), in 1999 and 2005, respectively. Currently, he is an associate professor at the Computer Architecture Department of the UPC and an associate researcher at Barcelona Supercomputing Center (BSC), where he leads the Energy-aware and Virtualization Technologies area within the Emerging Technologies for Artificial Intelligence group. His research interests are oriented towards green computing, virtualization/containerization, the smart management of resources in datacenters, and HPC/BD/AI convergence. He has been involved in several EU and industrial Research and Development projects.