The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Performance Comparison of Multi-container Deployment Schemes for HPC Workloads: An Empirical Study

Peini Liu · Jordi Guitart

Received: date / Accepted: date

Abstract The High-Performance Computing (HPC) community has recently started to use containerization to obtain fast, customized, portable, flexible, and reproducible deployments of their workloads. Previous work showed that deploying an HPC workload into a single container can keep bare-metal performance. However, there is a lack of research on multi-container deployments that partition the processes belonging to each application into different containers. Partitioning HPC applications have shown to improve their performance on virtual machines by allowing them to be set affinity to a NUMA (Non-Uniform Memory Access) domain. Consequently, it is essential to understand the performance implications of distinct multi-container deployment schemes for HPC workloads, focusing on the impact of the container granularity and its combination with processor and memory affinity. This paper presents a systematic performance comparison and analysis of multi-container deployment schemes for HPC workloads on a single-node platform, which considers different containerization technologies (including Docker and Singularity), two different platform architectures (UMA and NUMA), and two application subscription modes (exactly-subscription and over-subscription). Our results indicate that finer-grained multi-container deployments, on one side, can benefit the performance of some applications with low inter-process communication, especially in over-subscribed scenarios and when combined with affinity but, on the other side, they can incur some performance degradation for communication-intensive applications when using containerization technologies that deploy isolated network namespaces.

Peini Liu, Jordi Guitart
Computer Science Department, Barcelona Supercomputing Center (BSC), Barcelona, Spain
Computer Architecture Department, Universitat Politecnica de Catalunya (UPC), Barcelona, Spain
Tel: +34 - 93 405 40 47
Fax: +34 - 93 401 70 55
E-mail: peini.liu@bsc.es, jordi.guitart@bsc.es

This is a post-peer-review, pre-copyedit version of an article published in *Journal of supercomputing*. The final authenticated version is available online at: http://dx.doi.org/10.1007/s11227-020-03518-1

Keywords Docker · Singularity · Performance analysis · Deployment schemes · Multi-container · HPC workloads

1 Introduction

Modern computing infrastructure is evolving at a fast pace from using dedicated physical datacenters to cloud computing services. Virtualization, as a fundamental technology for cloud computing, allows efficient utilization and easy maintenance of the infrastructure. So far, this attractive paradigm has been widely used by leading commercial companies and communities to manage their clusters [16, 32]. The HPC community is also involved in this transformation of adopting virtualization to benefit from some of its well-known advantages [49], such as the encapsulation of specific software environments for each user, which allows for customization, portability, and research reproducibility [19]; 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 [9].

Virtualization was initially adopted in the form of hardware virtualization, which adds a layer of software between the operating system and hardware (so-called hypervisor), as well as an extra operating system for the guest. Historically, this incurred noticeable performance penalties, which have been dramatically reduced with the latest advances in virtualization. In particular, HPC workloads have taken advantage of the ability to leverage compute accelerators such as Graphics Processing Units (GPUs) from the virtual machines, or the ability to map the physical resources directly to the virtual machines. Furthermore, innovative deployment schemes have been also proposed to deal with the performance bottlenecks of virtualized HPC workloads in typical HPC architectures, such as multi-socket multi-core systems, like partitioning HPC applications into several virtual machines to prevent them spanning multiple NUMA domains [15]. This allows enabling affinity to a NUMA domain, which can enhance data locality in the L3 cache and reduce the RAM memory latency by preventing accesses to remote domains. Leveraging processor affinity as well can prevent process preemption and also enhance data locality in the L1 and L2 caches. Apart from exploiting data locality, partitioning can also optimize the packing of virtual machines and hence increase the utilization of the hosts since small-sized tasks can be allocated more easily without blocking in a waiting queue [11]. It can be also helpful to enhance the fault tolerance of the application, by replicating specific vital processes in separate virtual machines. If one of them fails, the replica can take over without downtime. Similarly, specific tasks of the application can also be checkpointed and recovered in case of a failure.

However, the still existing performance degradation of hardware virtualization regarding bare-metal executions [38] might not be acceptable for some HPC users. The emergence of containerization can alleviate that performance

gap [9, 41], as each container shares the underlying host kernel for OS services such as libraries, modules, and kernel functions. Therefore, a systematic analysis of HPC workloads running on containerized environments is necessary to understand the performance implications of using container technologies for deploying HPC workloads [8, 49], and to determine if the partitioning deployment schemes using multiple instances and the performance optimization methods based on affinity used for virtual machines are also appropriate with containers and what potential problems they might incur.

Performance analysis of HPC applications in containerized environments is an ongoing research problem [2, 35, 46]. Most related works evaluate single-container deployments and emphasize the possibility that deploying a HPC workload into a single container can achieve native performance [9, 41]. However, there is a lack of research on multi-container deployment solutions for a single-tenant multi-process HPC workload. Unlike the multi-container deployments holding workloads for multiple tenants [16–18, 24], using multiple containers to package a single-tenant multi-process/thread HPC workload refers to partitioning the processes or threads belonging to each application into different containers, obtaining in that way a finer-grained deployment. Whereas few works include experiments with different container granularity [8, 34, 35], 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 hardware platforms. To better identify optimal containerized deployment schemes and potential performance bottlenecks in a single multi-socket multi-core system, a proper and in-depth performance analysis is important before migrating HPC applications to containerized environments.

Early containerization implementations for deploying HPC benchmarks were mainly Linux-VServer, OpenVZ, and LXC [46]. However, containerization technologies have been evolving and Docker[1] has become the most used containerization software [3, 8, 35]. Docker provides an easy way to isolate the network and limit the resource usage of the containers, but some challenges remain with this technology to guarantee security and ensure performance when employed in HPC. Singularity[2], a novel HPC-oriented containerization technology, offers promising solutions for these issues [2, 20]. 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 in a natural way (without namespaces isolation). This work focuses on these two containerization technologies for deploying HPC workloads.

This paper presents a systematic performance comparison and analysis of containerized deployment schemes for HPC workloads. We address the next research questions: i) What is the impact of container granularity on the performance of multi-container deployment schemes for HPC workloads? ii)

[1] https://www.docker.com/
[2] https://sylabs.io/

What is the impact of processor and memory affinity on the performance of multi-container deployment schemes for HPC workloads?

Consequently, this paper contributes with a performance comparison and analysis of distinct multi-container deployment schemes for HPC workloads comprising i) different containerization technologies, ii) different container granularity, iii) different processor and memory affinity configurations, iv) different hardware platform settings, v) different application subscription modes.

The remaining of the paper is structured as follows. Related works are presented in Section 2. Section 3 describes the technologies behind containerization, especially Docker and Singularity, and also discusses the HPC benchmarks used in the evaluation. In Section 4, we evaluate the impact of container granularity on multi-container deployments, including also a comparison of different application subscription modes, different containerization technologies, and different hardware platform settings. Section 5 evaluates the impact of processor and memory affinity on multi-container deployments. Finally, Section 6 concludes the paper and discusses the future work.

2 Related Work

In order to improve the performance of HPC applications on virtualized multi-socket architectures, several works have proposed partitioning the HPC applications into several virtual machines to prevent them spanning multiple NUMA domains [14, 15]. The same idea of sizing virtual machines conforming with NUMA boundaries for throughput workloads (i.e. MPI jobs without communication) has been suggested by VMware in their reference architecture for virtualizing High Performance Computing [43].

Consequently, application partitioning comes together with the need to schedule the resulting virtual machines in the NUMA platform so that each of them optimizes its memory access locality [7, 31] or the access to local I/O devices [5]. Cheng et al. [7] presented a user-level scheduler that periodically adjusts the placement of virtual machines aiming for local node execution, that is, the VCPUs of a virtual machine are running on one NUMA node and its memory is also located on the same NUMA node. Rao et al. [31] proposed a load balancing algorithm to determine the optimal VCPU-to-core assignment by dynamically migrating VCPUs to minimize the penalty to access the uncore memory subsystem.

However, there is few empirical research yet evidencing whether the experience of virtualization can be applied with containerization with the same benefit for HPC applications. Yang et al. [48] proposed a management service for Docker containers based on OpenStack, which features a NUMA-aware mechanism that limits the accessible CPU and memory of containers to the same NUMA node. However, this work does not consider multi-container deployments from a single tenant but a simple scenario with two containers from two different tenants.

Several studies have compared the overhead of using virtualization and containerization technologies for HPC applications [6, 38, 40]. They claimed that containerization has less overhead than virtualization in most cases. As a result, many works have focused on the performance analysis of containerized deployments for HPC applications. Xavier et al. [46] firstly did in 2013 a full performance comparison of container-based technologies relevant at that time, mainly Linux-VServer, OpenVZ, and LXC, for HPC workloads. However, containerization technologies have evolved considerably since then, and new ones must be also evaluated for deploying HPC applications. In particular, Docker has become the most popular containerization software, and Singularity is also widely used in the community to support HPC workloads [2, 33, 44, 49]. These works have focused on evaluating the performance of an HPC application from a single-tenant on a single container allocated on a single node with different containerized technologies. Other works considered multi-tenant HPC workloads. Maliszewski et al. [24] investigated the performance of scientific workloads with single or multi-tenant instances in a single node, where each tenant held its independent application among other tenants. Jha et al. have studied HPC microservices in different container environments [17, 18]. Their work includes flexible deployments for HPC applications on a single node, from running a single or multiple applications in a single container, to running multiple containers each holding a single application. Whereas these studies above considered one or multiple tenants, co-located independent applications on a single node, or allocated one or more applications into a container, none of them considered a deployment scheme partitioning one application into several containers.

Other works have evaluated HPC workloads in distributed containerized platforms. Saha et al. [35] evaluated the performance of an HPC application running with several processes distributed across multiple containers using Docker Swarm, and studied different network methods, number of hosts, and ranks per container. Some of their results showed that one rank per container had degradation, but they did not explain in-depth why this occurred. In another work [34], the same authors presented a framework combining Apache Mesos and Docker Swarm which can orchestrate distributed containers with MPI processes across the nodes. They studied the overhead of running a different number of MPI processes and nodes, and presented a co-scheduling policy. These works showed that there is a possibility that distributed containers with partitioned processes from a single HPC application can be allocated on the same host. However, they mainly focused on the overhead of the orchestrator and the number of nodes, and did not study the specific interference among these containers while being allocated on the same node.

Chung et al. [8] considered the container granularity. Their work studies the scalability of running an HPC application on one or more containers. However, they only measured Docker performance for computation and data access intensive HPC applications, and did not distinguish the different subscription modes of the application or compare different containerization technologies.

None of these works study the joint impact of container granularity and processor and memory affinity settings for multi-container deployments, as we do in this work. Furthermore, none of them feature either an in-depth performance evaluation of Singularity, including its instance-based variant, and also a scenario adding CPU cgroups to its original implementation.

3 Background

This section introduces the technologies behind containerization, especially Docker and Singularity, and also discusses the benchmarks that we use to evaluate the performance of those technologies with HPC workloads.

3.1 Containerization technologies

3.1.1 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. Also, Docker 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) which are implemented by means of namespaces. Regarding resource limitation, some sets of dedicated resources that are defined by means of cgroups can be allocated to the Docker containers.

3.1.2 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 [2]. 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 in a natural way (without namespaces isolation). Because of this feature, the integration between Singularity and MPI can be transparent to the user. Users only need to run `mpirun` command as they run it on bare-metal machines, then the MPI process management daemon (ORTED) will handle the containers execution and the processes launching and communications. These make Singularity a first-class choice for HPC and scientific simulations [20, 36].

In late 2018, Singularity 3.0 was released [13]. This version brings a new functionality (so-called instances) to run containers in "daemon" mode, which

allows running them as services in the background. Singularity instances can have isolated network resources, and they also support cgroups functionality to restrict the resource usage. MPI applications can run in Singularity instances as if they were running in separated hosts, having its own network identity and using a SSH backend service to communicate. In this sense, Singularity instances somehow mimic Docker, while still keeping the advantages regarding security, thus we also include them as a part of our evaluation.

3.2 HPC Challenge Benchmark

The HPC Challenge (HPCC) benchmark suite[3] is widely used to evaluate the performance of HPC systems. Its design goal is to enable complete understandings of the performance characteristics of platforms [22]. 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. b_eff, PTRANS, FFT) and latency (e.g. b_eff, RandomAccess, FFT) [47]. We use common and standard units to evaluate the results of HPCC. The benchmarks are described as follows:

- **EP-DGEMM (DGEMM)** [22]: Real-valued dense matrix multiplication in double precision. Measures the floating point rate of execution in GFLOP/s.
- **G-FFT (FFT)** [1]: Global discrete Fast Fourier Transform of a vector. Measures the floating point rate of execution in GFLOP/s.
- **G-PTRANS (PTRANS)** [1]: Global Parallel matrix transpose. Exercises the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communication capacity (in GB/s) of the network.
- **EP-STREAM (STREAM)** [12]: Measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernels.
- **G-RamdomAccess (RA)** [1]: Random memory access. Measures the rate of integer random updates of memory (in GUP/s, i.e., GigaUpdates per second).
- **b_eff** [12]: Measures the latency (in microseconds) and bandwidth (in GB/s) of various communication patterns including ping-pong and ring.

3.2.1 Profiling analysis of the HPCC benchmarks

Due to the different attributes of each benchmark, some profiling of these benchmarks is useful for understanding their different MPI usage patterns [12],

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

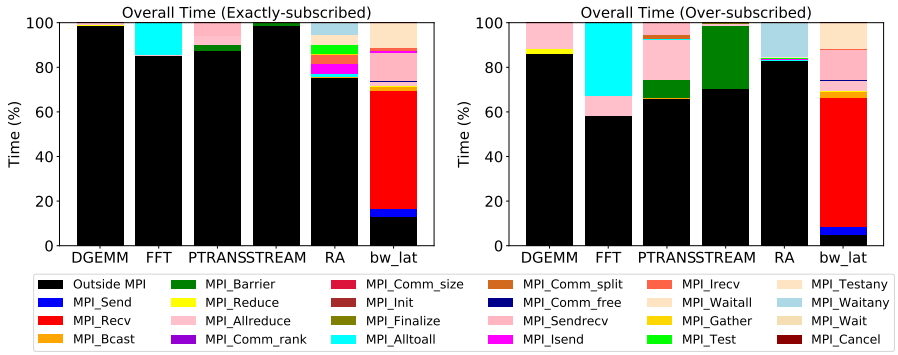

Fig. 1 HPCC MPI Profiling Analysis.

and can be used as a baseline for comparison with container-based executions in the evaluation.

Our analysis considers two different application subscription patterns, namely exactly-subscribed mode and over-subscribed mode. In the exactly-subscribed mode, the number of running processes is equal to the number of available processors. In the over-subscribed mode, there are more processes running than processors available, that is, it permits resource over-subscription. Tasks enabling over-subscription can obtain their resources sooner and decrease the waiting time, thus can be started earlier than in the exclusive mode [39].

Environment and Settings: The hardware platform consists of a single host with 2 x Intel 2697v4 CPUs (18 cores each, hyperthreading disabled), 256 GB RAM, 60 TB GPFS file system, and 1 Gb Ethernet network. The operating system on each server is CentOS 7.6. OpenMPI v4.0.3rc3 and HPCC benchmarks v1.5.0 are compiled by the GNU compiler collection in version 5.5.0. All the benchmarks are running with 16 processes on bare-metal. Exactly-subscribed mode is using 16 cores (8 from each socket). For over-subscribed mode, the over-commitment ratio is set to 2, which means using 8 cores (4 cores from each socket). We use an open source analysis tool Paraver[4] to profile MPI usage patterns of the benchmarks [30].

Results: Figure 1 shows the HPCC MPI profiling results. Segments of different colors correspond to the time spent within the various MPI functions with respect to the overall execution time. Table 1 presents the detailed time consumption percentages and the number of invocations of these MPI functions, which are classified according to the corresponding communication patterns.

From these results, we can divide these benchmarks broadly into two categories: MPI communication workloads, where processes need to communicate (frequently) with each other, and MPI throughput workloads, where there is (almost) no communication between processes [43]. Benchmarks whose name starts with G- and b_eff benchmark belong to the first category while others starting with EP- belong to the second one. Within the MPI communication

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

Table 1 HPCC Benchmark Profiling Analysis For Exactly- and Over-subscribed Mode.

Benchmark	Point to point communications				Collective communications				
	blocking ping-pong transfer	blocking concurrent transfer	non-blocking transfer	non-blocking synchronize	barrier synchronize	data movement			global reduce
	MPI_Send MPI_Recv	MPI _Sendrecv	MPI_Isend MPI_Irecv	MPI_Wait (any/all) MPI_Test (any/all) MPI_Cancel	MPI_Barrier	MPI _Alltoall	MPI _Bcast	MPI _Gather	MPI_(All) Reduce
DGEMM	E:<0.01% O:<0.01% NI:210				E:<0.01% O:<0.01% NI:16		E:0.01% O:0.01% NI:96		E:1.41% O:13.88% NI:96
FFT	E:<0.01% O:<0.01% NI:210				E:<0.01% O:0.02% NI:16	**E:14.22% O:32.65% NI:96**	E:<0.01% O:0.01% NI:96		E:0.56% O:9% NI:48
PTRANS	E:<0.01% O:<0.01% NI:220	**E:5.69% O:5.24% NI:480**			E:2.52% O:8.43% NI:96		E:<0.01% O:<0.01% NI:80		E:4.26% O:18.09% NI:592
STREAM	E:<0.01% O:<0.01% NI:210				E:1.25% O:28.42% NI:1296		E:<0.01% O:<0.01% NI:144	E:0.05% O:0.13% NI:16	E:0.05% O:0.92% NI:256
RA	E:<0.01% O:<0.01% NI:210		**E:8.89% O:0.62% NI:3276632**	**E:14.08% O:16.07% NI:3276344**	E:0.42% O:0.03% NI:4096	E:0.99% O:0.06% NI:4048	E:<0.01% O:<0.01% NI:96		E:<0.01% O:<0.01% NI:304
b_eff ping-pong	**E:56.32% O:61.6% NI:54404**				E:<0.01% O:0.01% NI:16		E:2.1% O:2.53% NI:6672		E:2.36% O:5.22% NI:9504
b_eff ring		**E:12.41% O:13.33% NI:50784**	**E:2.67% O:0.91% NI:101568**	**E:11.11% O:11.48% NI:25392**					

(E) and (O) means Exactly- and Over-subscribed Mode, respectively.
NI means Number of Invocations.

workloads, b_eff presents different patterns of point-to-point communications (e.g. blocking ping-pong transfer, blocking concurrent transfer, and non-blocking communications), which are also shown in G-PTRANS (blocking concurrent transfer) and G-RandomAccess (non-blocking communications). G-FFT uses mainly collective all-to-all communication. Thereby, all of these benchmarks can be used to evaluate the different aspects of interprocess communication. On the other side, MPI throughput workloads EP-STREAM and EP-DGEMM can be used to assess the memory bandwidth and the computation performance of the system, respectively. Note that our classification matches with the existing literature on HPCC [23], which has identified b_eff, G-RandomAccess, G-PTRANS, and G-FFT as performance-sensitive to the interconnection latency and/or bandwidth, whereas EP-DGEMM and EP-STREAM have been characterized as not sensitive to them.

4 Performance Analysis of Multi-container Deployments

4.1 Objective

In this section, we present an empirical performance evaluation of multi-container deployments of HPCC benchmarks with different container granularity. We evaluate different scenarios where we partition each application among an increasing number of containers, but decreasing number of processes per container (i.e., finer-grained container granularity). Within this evaluation, we consider different subscription modes on the application layer (exactly-subscription and over-subscription), different containerization technologies (including Docker and Singularity), and different hardware platform settings (UMA and NUMA).

4.2 Method

The idea of containerization is to provide a pool of resources for a group of processes/threads. However, the grouping of the processes/threads within a job admits several combinations, as well as the resource group provided by the hardware can also vary. Thus, the impact of containerized deployments can be analyzed by changing the elements at both ends of the mapping.

The container-based deployment model for HPC workloads is shown in Figure 2. It consists of three modules: a job contains several processes/threads, which can be divided into groups of various sizes and are packaged into different containers; a host has multiple resources organized into sets which are able to run the containers; and a containerization layer including containers that holds the mapping between a group of processes/threads and a set of resources. Our evaluation compares and analyzes the performance of HPC workloads by considering deployment schemes with different number of containers and processes per container and using different containerization technologies on the containerized layer.

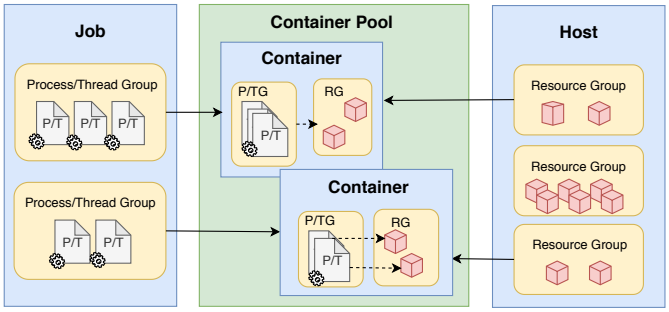

Fig. 2 Container-based deployment model for HPC workloads.

4.3 Experimental setup

This section describes the experimental setup used for performance evaluation. All the results of each experiment are derived from the average of 10 executions and the bare-metal executions are considered as baselines of every scenario. We perform unpaired two-samples T tests to assess whether the performance difference between the means in our experiments is statistically significant or due to randomness. We consider that a P-value lower than threshold 0.05 denotes a statistically significant difference.

Classical unpaired two-samples T tests require that the two groups of samples are normally distributed, so we first verify that by using Shapiro-Wilk tests [37]. When some of the groups of samples being compared are not normally distributed, we use Mann-Whitney tests [25] instead of the classical two-samples T tests. Unpaired two-samples T tests also require that the variances of the two groups are equal. We verify this by using Fisher's F-tests. When the variances are not equal, we use Welch T tests [45] instead of the classical T tests.

Environment: Our experiments are executed on a single-host HPC platform. The hardware characteristics of this host have been described in Section 3.2.1. Figure 3 shows a schematic view of its architecture. There are two sockets containing 18 cores each. The distance for accessing local and remote memory is 10 and 21, respectively. Each core has its own L1 and L2 cache, and L3 cache is shared by the 18 cores in the same socket. We use this host to define two different hardware platform settings: one with Non-Uniform Memory Access (NUMA) and another with Uniform Memory Access (UMA). Table 2 summarizes the hardware characteristics of these two settings. Both of them have the same number of cores, each one with L1 data cache (32K), L1 instruction cache (32K), and L2 cache (256K). In the NUMA hardware setting, those cores belong to 2 different sockets, each one with its own L3 cache (45MB); in the UMA hardware setting, the cores all belong to a single socket with a single 45MB L3 cache. The software stack for both host and containers, and its compilation environment are described in Table 3.

Benchmark settings: The settings for HPCC are the same as described in Section 3.2.1, so all the benchmarks are running with 16 MPI processes in

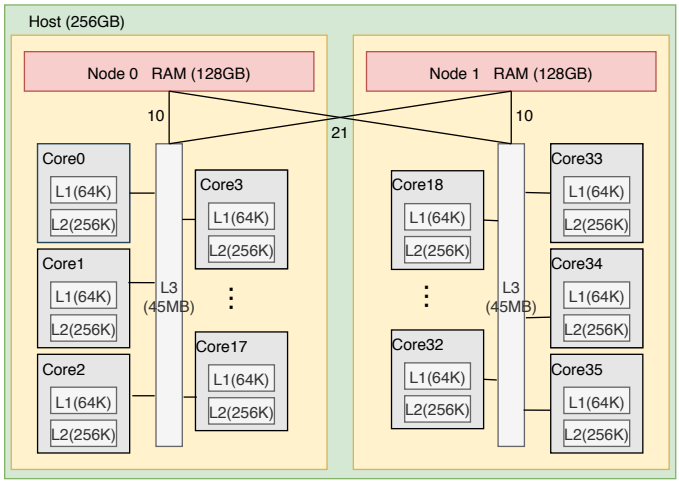

Fig. 3 A schematic view of our single-host HPC platform with two sockets and 18 cores per socket with shared L3 cache.

Table 2 Overview of the multi-socket multi-core hardware settings used in the experiments.

Hardware Setting	#sockets	#cores	L3(MB)	RAM(GB)
NUMA	2	16(8 per socket)	90(45 per socket)	256(128 per socket)
UMA	1	16	45	128

Table 3 Software Stack.

Software	Version	Location	Compiler
Linux	CentOS 7.6.1810	Host & Container	
Docker	19.03.5	Host	
Singularity	3.5.1	Host	
OpenMPI	OpenMPI-4.0.3rc3	Host & Container	GCC 5.5.0

all the scenarios. In the exactly-subscribed mode, those processes run on 16 cores, whereas in the over-subscribed mode they run on 8 cores. Additionally, we enable OpenMPI MCA parameter `mpi_yield_when_idle` for all the over-subscribed scenarios to prevent the degradation from the OpenMPI (see Section 4.4.4).

Container granularity settings: Different deployment schemes for evaluating container granularity are presented in Figure 4. Figure 4 (a) presents the settings of deployment scenarios on the NUMA hardware platform setting and Figure 4 (b) on the UMA hardware platform setting. E or O refers to the application running on exactly-subscribed mode or over-subscribed mode, respectively. E1-E5 and O1-O4 reflect the different granularity of the containers. As shown in Table 4, E1 and O1 use a single container, while E2-E5, O2-O4 are scenarios with an increasing number of containers, but decreasing number of processes per container.

In the experiments denoted as 'ANY', each container could use any of the available cores according to the used hardware platform setting (see Table

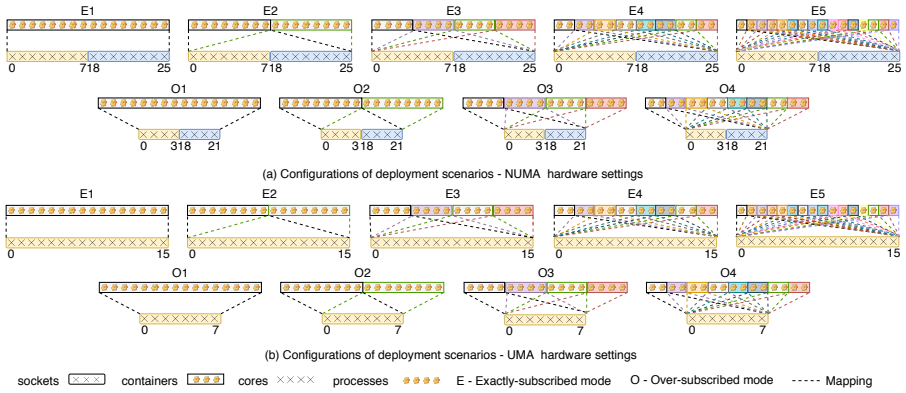

Fig. 4 Containerized deployment scenarios.

Table 4 Settings for containerized deployment scenarios.

Scenarios	#containers	#processes per container	Used cores and sockets
E1,E2,E3,E4,E5	1,2,4,8,16	16,8,4,2,1	NUMA: cores 0-7,18-25; sockets 0-1 UMA: cores 0-15; socket 0
O1,O2,O3,O4	1,2,4,8	16,8,4,2	NUMA: cores 0-3,18-21; sockets 0-1 UMA: cores 0-7; socket 0

4). The actual distribution of the running processes on the available cores is decided dynamically by the CFS Linux scheduler. In the experiments denoted as 'PIN', we enforce a 1-to-1 binding from the processes of the application to the available cores according to the used hardware platform settings. This binding holds during the entire execution of the application. We include this setting in the comparison to serve as a reference and to assess the performance reproducibility when variable process placement is eliminated, especially with over-subscription.

Containerization technologies: Docker and Singularity containerization technologies are evaluated in this work. We include also some variants of Singularity in the comparison. The different features of these technologies are described in Table 5: 1) *Docker*: Docker containers run isolated into different namespaces and cgroups. 2) *Singularity*: Default Singularity version, which executes the containers like they were native programs or scripts on a host computer, without encapsulating them on separated namespaces or cgroups. 3) *Singularity-instance*: Similar to Docker, Singularity container instances, which are persistent versions of the container image, run isolated in the background into different namespaces and cgroups. 4) *Singularity+cgroup*: Plain Singularity containers are executed (not instances), but each of them runs on its own cpu cgroup. Note that this cgroup should be a hierarchical cgroup, otherwise the scheduler will allocate resources among multiple root level cgroups thus bringing some degradation of performance.

Table 5 Features of different containerization technologies.

	Docker	Singularity (instance)	Singularity	Singularity (+cgroup)
Namespaces	Yes	Yes	No	No
Cgroup	Yes	Yes	No	Yes
File System	overlay	ext3	ext3	ext3
Network	Bridge	Bridge	Host	Host

Performance analysis tools: We use Paraver to profile MPI usage patterns of the benchmarks. We capture also performance event counters (through Perf[5]) and operating system metrics, such as context-switches, migrations, and memory accesses, from representative executions of the benchmarks and we use them to explain the obtained performance results.

4.4 Results

4.4.1 Impact of containerization technology and container granularity on multi-container deployments

In this section, we evaluate the performance impact of different granularity of containers with different containerization technologies through scenarios E1-E5 and O1-O4.

MPI communication workloads: As a result of the profiling analysis in Section 3.2.1, we concluded that b_eff (including Ping-Pong and Ring patterns), G-RandomAccess, G-PTRANS, and G-FFT benchmarks can be classified as MPI communication workloads. In particular, b_eff spends about 87% of overall runtime in MPI (95% when over-subscribed), G-RandomAccess spends about 24% of overall runtime in MPI (16% when over-subscribed), G-PTRANS spends about 13% of overall runtime in MPI (34% when over-subscribed), and G-FFT spends about 15% of overall runtime in MPI (42% when over-subscribed).

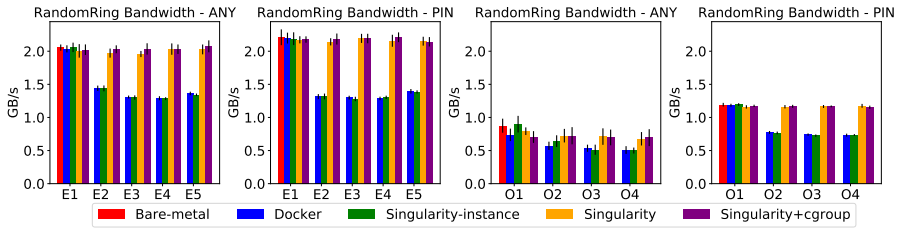

Fig. 5 Impact of container granularity in b_eff(RandomRing) bandwidth on NUMA hardware platform setting.

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

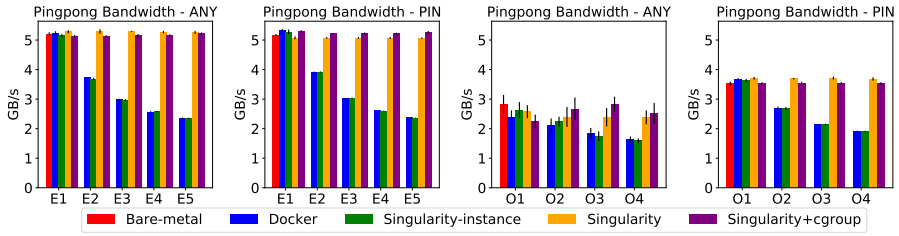

Fig. 6 Impact of container granularity in b_eff(PingPong) bandwidth on NUMA hardware platform setting.

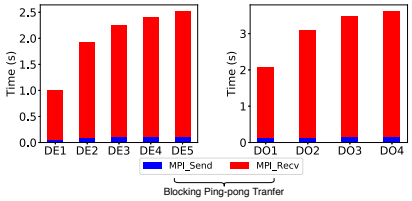

Fig. 7 Time spent in MPI communication patterns of b_eff(PingPong) benchmark for PIN scenarios on Docker (NUMA hardware platform setting).

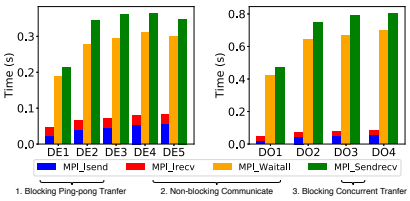

Fig. 8 Time spent in MPI communication patterns of b_eff(Ring) benchmark for PIN scenarios on Docker (NUMA hardware platform setting).

Figures 5 and 6 show the bandwidth results for the b_eff benchmark, differentiating PingPong and RandomRing MPI point-to-point communication patterns. We omitted the latency results as they were essentially following the same trend. There is significant performance degradation when the processes run on multiple containers in Docker and Singularity-instance (scenarios E2 to E5 and O2 to O4) regarding single-container deployments, as the P-values of the corresponding T-tests range from $4.9e^{-21}$ to $3.3e^{-11}$, which are all clearly lower than 0.05. In order to better understand this behavior, Figures 7 and 8 detail the time spent in seconds on each MPI function for the various communication patterns in this benchmark when running on Docker. This time is greater when running with multiple containers for blocking ping-pong transfer patterns (MPI_Send and MPI_Recv), around 90% on scenario E2, and non-blocking transfer patterns (MPI_Isend and MPI_Irecv), around 37% on scenario E2, and increases with the number of containers, +16% (E3), +7% (E4), +4% (E5) and +12% (E3), +8% (E4), +4% (E5), respectively. The time is also greater when running multiple containers for non-blocking synchronization, around 47% on scenario E2, and blocking concurrent transfer patterns, around 61% on scenario E2, but it barely increases with the number of containers, +7% (E3), +5% (E4), -3% (E5) and +4% (E3), +1% (E4), -4% (E5), respectively.

This degradation occurs because the processes running on separated containers in Docker and Singularity-instance are deployed on isolated network namespaces and have to use the TCP/IP network stack rather than shared-memory to communicate with one another. Executions with a single container or using Singularity do not have degradation on any of the scenarios when

comparing with bare-metal (the P-values of the corresponding T-tests range from 0.34 to 0.89, clearly higher than 0.05, thus the difference is not statistically significant) because the processes do not communicate through isolated network namespaces. All the processes belong to the same namespace and can use shared-memory to communicate as when running on bare-metal.

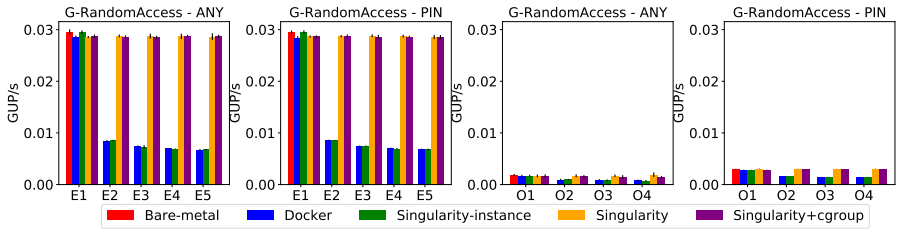

Fig. 9 Impact of container granularity in G-RandomAccess performance on NUMA hardware platform setting.

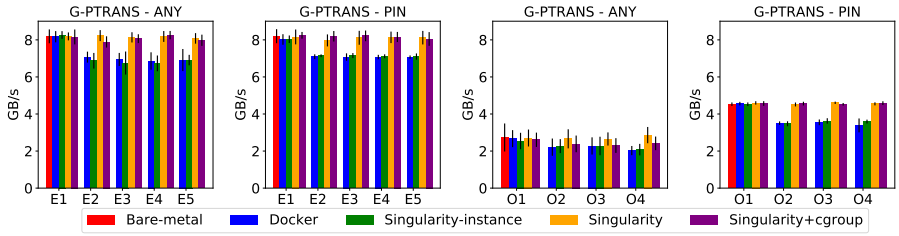

Fig. 10 Impact of container granularity in G-PTRANS performance on NUMA hardware platform setting.

According to the profiling analysis of benchmarks in Section 3.2.1, among the benchmarks classified as MPI communication workloads, G-RandomAccess and G-PTRANS present some MPI point-to-point communication patterns (see Figures 9 and 10). In particular, G-Randomaccess spends about 23% of overall runtime on point-to-point non-blocking communication (16% when over-subscribed). Thereby, Docker and Singularity-instance incur performance degradation (around 70% on scenario E2) that increases slightly with the number of containers (up to 77% on scenario E5). This degradation is statistically significant as the P-values of the corresponding T-tests are lower than 0.05 (ranging from $9.4e^{-22}$ to $1.8e^{-18}$). G-PTRANS spends about 5% of overall runtime on point-to-point blocking concurrent transfers (e.g. MPI_Sendrecv) (also 5% when over-subscribed), thus Docker and Singularity-instance degrade on running multiple containers due to using the network stack (around 15-17% degradation on scenarios E2 to E5, which is statistically significant as the P-values of the T-tests with respect to single-container deployments range from

$3.2e^{-7}$ to $1.8e^{-4}$, which are lower than 0.05), but this degradation does not increase with the number of containers (P-values of scenarios E3-E5 with respect to E2 range from 0.2 to 1, which are higher than 0.05). The performance degradation in G-PTRANS is significantly lower than b_eff and G-RandomAccess because the number of invocations to MPI functions is considerably lower. As before, Singularity and Singularity+cgroup does not incur any statistically significant degradation. This is confirmed in the corresponding T-tests where all the P-values are higher than 0.05 (ranging from 0.07 to 0.9).

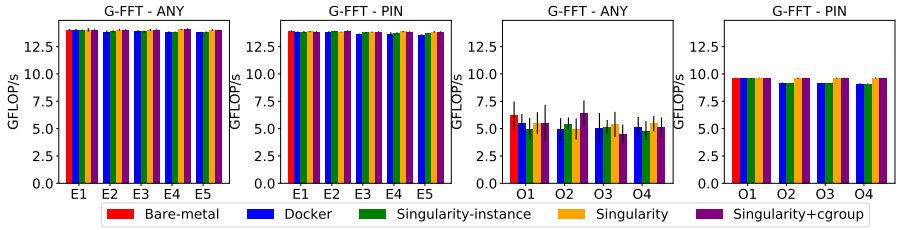

Fig. 11 Impact of container granularity in G-FFT performance on NUMA hardware platform setting.

Unlike previous MPI communication workloads, G-FFT mainly uses collective communication (mostly MPI_Alltoall) for data movement. From the results in Figure 11, the performance degradation when running multiple containers in Docker and Singularity-instance is almost negligible when exactly-subscribed (around 1%) and quite small when over-subscribed (around 4% when pinning processes). In both cases, the P-values of the corresponding T-tests show that those small differences are statistically significant and not due to randomness (ranging from $5.3e^{-6}$ to $7.5e^{-3}$). What makes the difference is the number of invocations of MPI calls. As a rule of thumb, applications doing point-to-point communication perform many more invocations to MPI functions than applications using collective communication. In particular, as shown in Table 1, G-FFT does only 210 point-to-point and 256 collective invocations (vs. more than 50000 point-to-point invocation in b_eff(PingPong)), hence the degradation is considerably lower.

MPI throughput workloads: The results of the profiling analysis in Section 3.2.1 allowed to classify EP-STREAM and EP-DGEMM benchmarks as MPI throughput workloads. As shown in Figures 12 and 13, which depict the performance of those benchmarks when running with various container grain sizes and containerization technologies, those workloads do not show significant performance variation regarding the baseline when increasing the number of containers per host. For instance, the P-values of the T-tests for multi-container deployments of EP-STREAM regarding single-container deployments range from 0.05 to 0.85 (higher than 0.05). This is due to the low amount of inter-process communication, namely 1.4% of overall runtime in MPI (29.5% when over-subscribed, but mostly in synchronization functions) for EP-STREAM,

and 1.5% of overall runtime in MPI (13.9% when over-subscribed, but mostly in the global reduce) for EP-DGEMM.

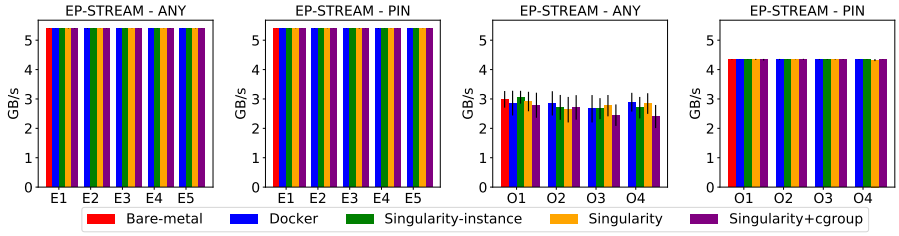

Fig. 12 Impact of container granularity in EP-STREAM performance on NUMA hardware platform setting.

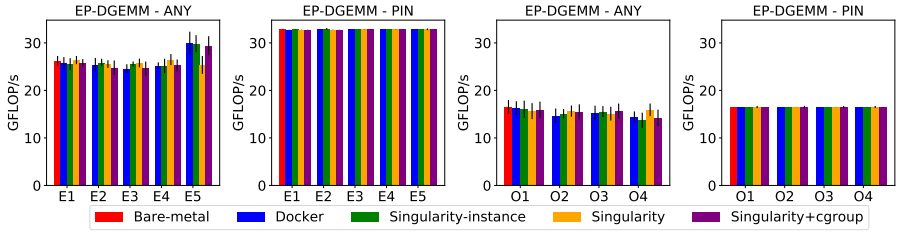

Fig. 13 Impact of container granularity in EP-DGEMM performance on NUMA hardware platform setting.

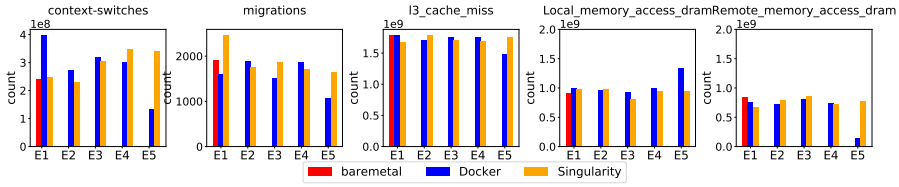

Fig. 14 Performance event counters of EP-DGEMM for ANY scenarios on NUMA hardware platform setting.

Something noticeable in Figure 13 is the performance improvement of EP-DGEMM in scenario ANY-E5 (which runs a single MPI process on each container) regarding the other deployment scenarios with all the containerization technologies but plain Singularity. In those technologies, ANY-E5 shows significant difference compared to ANY-E1, as the P-values of the corresponding T-tests range from $1.7e^{-3}$ to $7.3e^{-3}$ (lower than 0.05). As shown in Figure 14, which depicts relevant performance counters of EP-DGEMM, scenario E5

with Docker has considerably less process migrations and context-switches than the other deployment scenarios. It also shows better cache utilization (less L3 misses), more local memory accesses, and only minimal remote memory accesses. These are consequences of the scheduling of the containers (i.e. the 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 (although not so deterministic). The same occurs with Singularity-instance, but not with Singularity because all the processes run within the same cgroup.

Performance variability and impact of 1-to-1 process pinning: Most benchmarks (b_eff, G-PTRANS, G-FFT, EP-STREAM, and EP-DGEMM) present some performance variability in the ANY over-subscribed scenarios, which does not occur when binding processes to processors. In addition, EP-DGEMM also shows some variability in the ANY exactly-subscribed scenarios, which comes mainly from the process context-switches and migrations, and can be avoided again by pinning the processes. Apart from eliminating the performance variability, 1-to-1 process pinning also improves the performance on over-subscribed scenarios by eliminating the variable process placement. In the same way, it also improves the performance of EP-DGEMM when exactly-subscribed.

4.4.2 Impact of the cgroup scheduling on multi-container deployments

As shown in previous sections, Docker and Singularity-instance incurred significant performance degradation on MPI communication workloads when running multiple containers due to the interprocess communication between them. In this section, we assess whether other container-supporting technologies, such as cgroups, could be also contributing to that performance degradation.

Linux cgroups are mechanisms from kernel-level that control the resource allocation by restricting CPU, memory, network, etc., for each group of processes. One of them is the CPU controller, which is responsible for grouping tasks together that will be viewed by the scheduler as a single unit. The CFS (Completely Fair Scheduler) scheduler applies the principle of sharing the resources fairly among these groups at the same level of the hierarchy, which means it will first divide CPU time equally between all entities in the same level, and then proceed by doing the same in the next level [10].

To assess the impact of cgroups, we included the Singularity+cgroup experiments, which run each container in a separated CPU cgroup (as done by default by Docker and Singularity-instance), which means that each container will run their processes in their own group sharing the CPU time allocated.

As shown in previous figures, Singularity+cgroup achieves the same performance than Singularity for all the benchmarks in exactly-subscribed scenarios, but it incurs some performance degradation (similar to Docker and Singularity-instance) in some of the benchmarks on over-subscribed scenarios. For instance, this is especially noticeable with G-PTRANS on ANY scenarios O2, O3, and

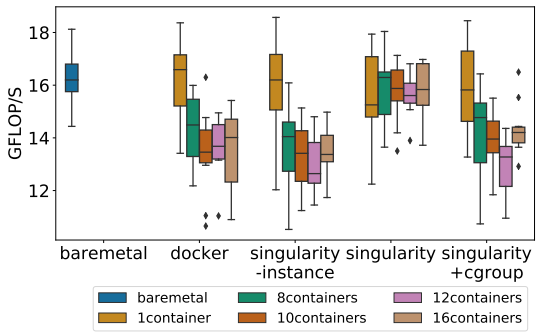

Fig. 15 Performance comparison of EP-DGEMM with different number of containers.

O4, and EP-DGEMM (and to a lesser extent on G-FFT) on ANY scenario O4. In those scenarios, the cgroup scheduling performed by the CFS results in imbalanced executions. CFS tries to maintain fair time allocation among cgroups, not processes, but it is not especially accurate tracking the load of scheduling entities when they are groups of processes (i.e. cgroups). Those coarse-grain load measurements are then used to calculate the load of the processors and decide about load balancing from busier to idler processors, resulting in an imbalanced allocation of processes to processors [4, 21]. This is critical in over-subscribed scenarios where processes must share processors efficiently to ensure progress.

This can be confirmed in Figure 15, which shows the EP-DGEMM performance on over-subscribed mode including additional scenarios that deploy a higher number of containers cgroups than the number of available CPUs. Whereas holding all the MPI processes in a single container provides bare-metal performance, having multi-container deployments causes significant performance degradation in all the containerization technologies except Singularity, as Singularity is not using distinct cgroups. The corresponding T-tests confirm that the P-values for Singularity are higher than 0.05 (ranging from 0.13 to 0.49), whereas for the other containerization technologies they are lower than 0.05 (ranging from $1.9e^{-6}$ to $7.4e^{-3}$).

4.4.3 Impact of the hardware platform setting on multi-container deployments

The hardware platform setting has also an impact on the performance of the different benchmarks, but this is mostly unrelated with the containerization technology and the deployment scheme. As such, in the UMA setting there is also significant performance degradation for MPI communication workloads when the processes run on multiple containers in Docker and Singularity-instance because the processes running on separated containers are deployed on isolated network namespaces. Executions with a single container, using Singularity, or for MPI throughput workloads do not have degradation on none of the scenarios when comparing with bare-metal.

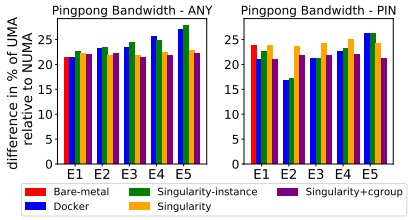

Fig. 16 Bandwidth difference (in %) of UMA relative to NUMA in b_eff(PingPong).

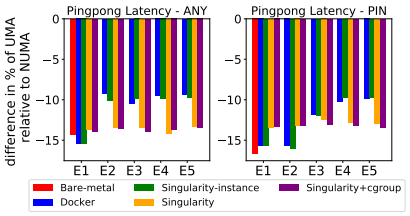

Fig. 17 Latency difference (in %) of UMA relative to NUMA on b_eff(PingPong).

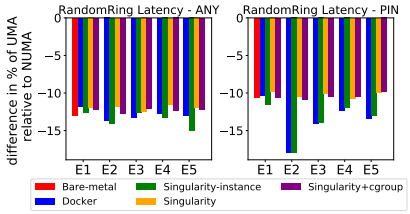

Fig. 18 Latency difference (in %) of UMA relative to NUMA on b_eff(RandomRing).

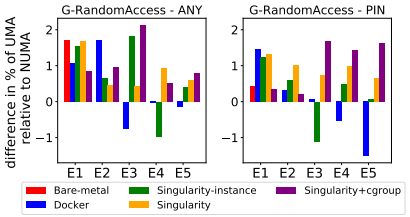

Fig. 19 Performance difference (in %) of UMA relative to NUMA on G-RandomAccess.

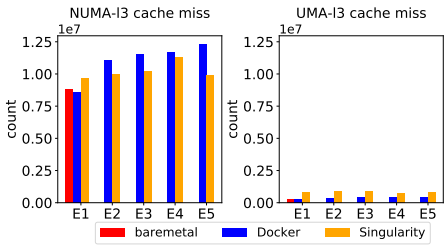

Fig. 20 Performance counters for b_eff(PingPong) benchmark on ANY scenarios.

The performance difference between the NUMA and UMA hardware platform settings depends on the specific characteristics of each benchmark. Figures 16-19, 21, and 23-26 present the performance difference (in %) of UMA relative to NUMA for each benchmark. The difference between these two hardware settings is that UMA optimizes the latency of accessing memory by improving the cache usage and eliminating the remote memory accesses, while NUMA optimizes the memory bandwidth. Given the performance variability in the ANY over-subscribed scenarios, which makes it difficult to obtain meaningful conclusions, we focus the comparison in this section in the exactly-subscribed scenarios.

As shown in Figures 16-19, PingPong Bandwidth/Latency and Ring Latency benchmarks show significant better performance in the UMA setting, ranging from 15% to 27%. In those benchmarks, the P-values of the T-tests comparing UMA and NUMA scenarios range from $1.0e^{-25}$ to $1.8e^{-4}$ (lower than 0.05).

G-Randomaccess benchmark also shows some improvement (less than 2%), but the results are inconclusive, as the P-values of the T-tests are higher than 0.05 for some scenarios (ranging from 0.06 to 0.94) and lower than 0.05 for others (ranging from $2.5e^{-3}$ to 0.04). The MPI processes on these benchmarks communicate through small-sized point-to-point messages and are not memory intensive. In the UMA setting, all the processes run in a single socket, sharing the L3 cache and the local memory, which enhances the use of the cache (less L3 misses as shown in Figure 20) and reduces the number of memory accesses regarding the NUMA setting.

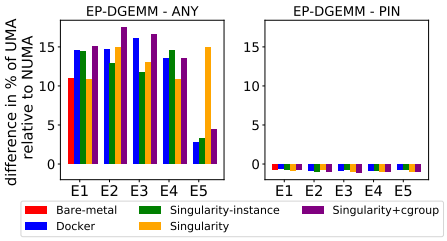

Fig. 21 Performance difference (in %) of UMA relative to NUMA on EP-DGEMM.

As shown in Figure 21, EP-DGEMM also performs better in the UMA setting for ANY scenarios. The improvement is significant as the P-values of the corresponding T-tests for scenarios E1-E4 and E5-Singularity range from $1.6e^{-9}$ to $4.5e^{-2}$ (lower than 0.05). The difference is less statistically significant in scenario E5 with the other containerization technologies as the corresponding P-values are around 0.14. Placing all the MPI processes in the same socket has resulted in better cache sharing and allows them to better access the local memory, which reduces the latency of accessing remote memory. As shown in Figure 22, EP-DGEMM in the UMA setting performs only local memory accesses, whereas it does a mixture of local (56% of the L3 cache misses count) and remote memory accesses in the NUMA setting. For PIN scenarios, EP-DGEMM in the NUMA setting already has good memory locality (local memory accesses count is 99% of L3 cache misses count), thus UMA does not bring any advantage on avoiding remote memory accesses latency, and hence the performance of EP-DGEMM on both hardware platform settings is almost the same, with NUMA bringing a small improvement around 1%-2%, which is statistically significant as the corresponding P-values range from $2.0e^{-15}$ to $2.8e^{-3}$.

As shown in Figures 23-26, EP-STREAM, G-FFT, b_eff(Ring Bandwidth), and G-PTRANS have significantly worse performance in the UMA setting. In particular, EP-STREAM is 48% slower in the UMA setting, with P-values of the T-tests ranging from $3.3e^{-36}$ to $1.8e^{-4}$ (lower than 0.05). As the processes are mostly accessing the local memory in the NUMA setting (99.6% for ANY scenarios and 99.9% for PIN scenarios), UMA cannot bring additional benefit by avoiding remote memory accesses, but introduces more memory contention

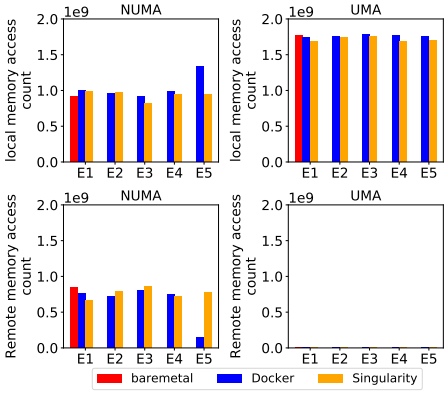

Fig. 22 Performance counters for EP-DGEMM benchmark on ANY scenarios.

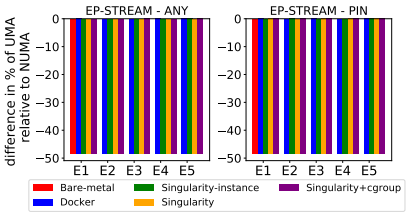

Fig. 23 Performance difference (in %) of UMA relative to NUMA on EP-STREAM.

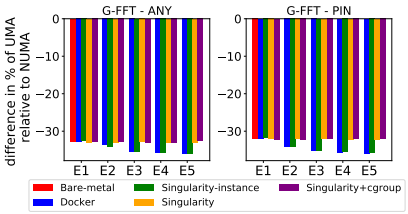

Fig. 24 Performance difference (in %) of UMA relative to NUMA on G-FFT.

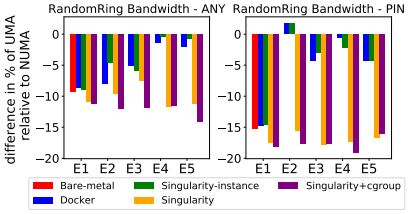

Fig. 25 Bandwidth difference (in %) of UMA relative to NUMA on b_eff(RandomRing).

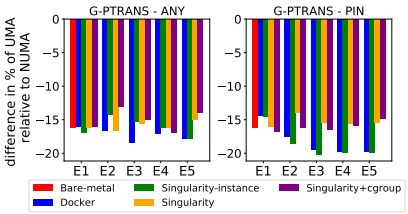

Fig. 26 Performance difference (in %) of UMA relative to NUMA on G-PTRANS.

because it has only one socket which reduces the available memory bandwidth. G-FFT, b_eff(Ring Bandwidth), and G-PTRANS communicate their processes using large messages. In a single host, their performance is also limited by the memory bandwidth, and for this reason, the NUMA setting provides better performance for them. For example, G-FFT is 32%-36% faster (P-values ranging from $7.5e^{-35}$ to $1.8e^{-4}$) and G-PTRANS is 14%-20% faster (P-values ranging from $1.0e^{-16}$-$2.8e^{-3}$).

In order to measure the memory contention that occurs on those benchmarks, we calculate the memory contention ratio among cores in the UMA and NUMA settings by using the model proposed by Tudor and Teo [42]. Same as those

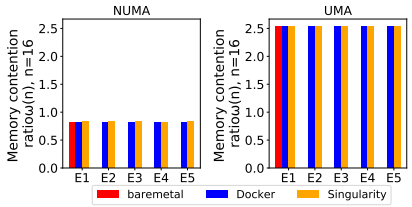

Fig. 27 Memory contention ratio for EP-STREAM benchmark on ANY scenarios.

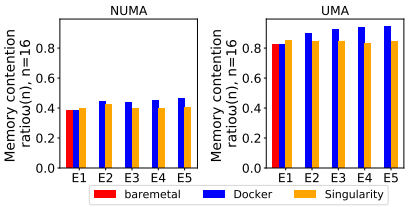

Fig. 28 Memory contention ratio for G-FFT benchmark on ANY scenarios.

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 (including stall cycles that are not due to resource contention). A higher ω means more memory contention. As shown in Figures 27 and 28, which depict the memory contention ratio for EP-STREAM and G-FFT, respectively, memory contention is higher in the UMA platform setting, because there are 16 processes concurrently accessing the local memory and the UMA platform setting cannot benefit from a second memory controller to serve their operations, which increases the contention in L3 cache and local memory.

4.4.4 Proper configuration of multi-container deployments with over-subscription

Unlike the exactly-subscribed mode where OpenMPI can run its message passing engine always in aggressive mode (never giving up the processors to other processes), over-subscribed mode requires the OpenMPI engine to run in degraded mode and frequently yielding the processor to its peers when idle, thereby allowing all processes to make progresses [28]. The awareness of the aggressive or degraded mode of OpenMPI engine is usually automatic, although the user can use the MCA parameter `mpi_yield_when_idle` to control whether an MPI process runs in aggressive or degraded performance mode [27].

However, when using containers to run an MPI application in over-subscribed mode, things are more complicated. The difference between the aggressive and degraded modes in the OpenMPI engine when running on containers can be observed in Figure 29. For bare-metal, Singularity, and single-container deployments of Docker and Singularity-instance, the performance of the 'default' configuration matches with the performance when `mpi_yield_when_idle` is enabled, as the OpenMPI engine can automatically detect the over-subscription and run in degraded mode. However, for scenarios with multiple containers of Docker and Singularity-instance, the degraded mode must be set explicitly by enabling `mpi_yield_when_idle` in the `mpirun` command in order to let the process yield the processor to its peers. Otherwise, MPI processes running in

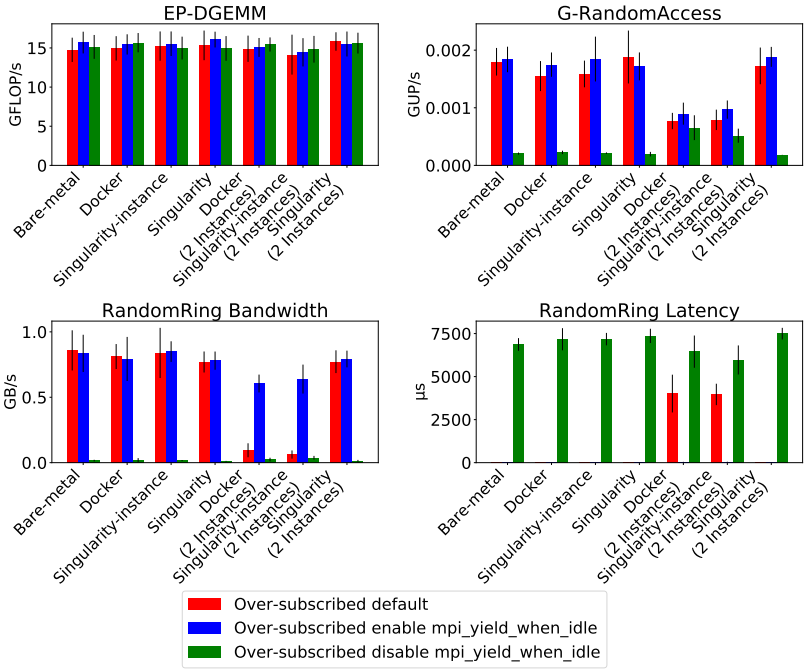

Fig. 29 Comparison of over-subscribed mode with different *mpi_yield_when_idle* configurations on single and multiple containers environments.

disparate containers are not aware of their peers and will not yield the processor, thus degrading the performance ('default' configuration is degraded as when `mpi_yield_when_idle` is disabled). The results also show that, the most time the MPI processes are blocked in the MPI library, the most noticeable the benefits of declaring degraded mode operation are, indicating that MPI communication workloads will be especially sensitive to this.

These observations can also be confirmed through unpaired two-samples T-tests between the 'default' configuration and the one with `mpi_yield_when_idle` enabled. In particular, for the EP-DGEMM benchmark, the P-values of the T-tests for all the scenarios are all higher than 0.05 (ranging from 0.16 to 0.74), hence both configurations have the same performance. On the other side, for the other three benchmarks, which are more communication intensive, the P-values of the T-tests on Docker and Singularity-instance with more than one instance are lower than 0.05 (ranging from $2.2e^{-12}$ to 0.02), hence the 'default' configuration provides statistically worse performance than enabling `mpi_yield_when_idle`.

4.4.5 Summary

To sum up, the findings from our previous evaluation of multi-container deployments are as follows:

- For Docker and Singularity-instance, multi-container deployments incur some performance degradation for MPI communication workloads, because the processes running on separated containers are deployed on isolated network namespaces and have to use the TCP/IP network stack rather than shared-memory to communicate with one another. This could be avoided by enabling shared-memory among the distinct containers and making the MPI engine aware of that shared-memory area [15].
- Singularity has close to bare-metal performance because the containers share the network and IPC namespaces and can use shared-memory to communicate the processes.
- Multi-container deployments of MPI throughput workloads do not incur significant performance degradation regarding bare-metal when increasing the number of containers due to the low amount of interprocess communication. Finer-grained deployments show a performance improvement because they simplify the scheduling in a similar way to when processes are pinned explicitly, which encourages further study of the impact of affinity on performance (see next section).
- On over-subscribed mode, some performance degradation is due to the scheduling of cgroups by Linux CFS, which results in an imbalanced allocation of processes to processors, because CFS is not especially accurate tracking the load of scheduling entities when they are groups of processes (i.e. cgroups).
- The advantage/disadvantage of using the UMA hardware platform setting is not directly related with any containerization technology or container granularity, but with the application and hardware setting characteristics, such as the cache usage or the memory bandwidth. In particular, applications with low memory bandwidth requirements and good data locality perform better in the UMA setting, while memory-intensive applications perform better in the NUMA setting.
- It is necessary to enable MCA parameter `mpi_yield_when_idle` for multi-container deployments on over-subscribed mode, especially with MPI communication workloads, because this enables MPI processes on different containers to run in degraded mode.

5 Performance Analysis of Multi-container Deployments with Processor and Memory Affinity

5.1 Objective

As mentioned in the introduction, Ibrahim et al. [15] have shown that the performance of HPC workloads on multi-socket NUMA architectures degrades when virtual machines span several NUMA domains. They claimed that the degradation was caused by the two-level memory management inherent in virtualized systems combined with the lazy page reclamation policies implemented in modern kernels. Our results in the previous section showed that the

performance of HPC workloads running on a single container does not suffer such degradation when spanning several NUMA domains, basically because containers use only one-level memory management (the same as bare-metal processes). Consequently, partitioning HPC workloads into multiple containers and containing each one in a single NUMA domain through affinity is not expected to show noticeable benefits from a memory translation perspective for most of the benchmarks analyzed in this paper.

Nevertheless, the impact of container granularity on multi-container deployments with affinity can be significant depending on the CPU and memory usage characteristics of each benchmark. For example, restricting the range of possible 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.

In this section, we evaluate the impact of setting affinity on partitioned workloads when using containers, by assessing the performance of multi-container HPC applications with different processor and memory affinity configurations. In particular, we test different scenarios where we partition each application among an increasing number of containers but decreasing number of processes per container, and we configure each container with some affinity settings, which include i) affinity of the container to a set of cores from two sockets and to the corresponding local and remote memory nodes (i.e. CPU), ii) affinity of the container to a set of cores from a single socket and to the local memory node (i.e. CPUMEM), and iii) 1-to-1 affinity of the processes of the container to cores from a single socket and to the local memory node (i.e. CPUMEMPIN). Within this evaluation, we consider different subscription modes on the application layer (exactly-subscription and over-subscription), different containerization technologies (including Docker and Singularity), and different hardware platform settings (UMA and NUMA).

5.2 Method

Most containerization technologies use by default the namespace capability of the control groups, but utilize the resource control capability only when the user explicitly provides the parameters [29]. For example, considering the experiments in Section 4, from the application perspective, the workload is partitioned into several containers. However, from the kernel perspective, all of them are still sharing the same resources in the system (and competing for them). Thereby, the kernel has to arbitrate this competition to access the system hardware or software resources and multiplex the containers to ensure that all of them receive a fair share.

The purpose of processor and memory affinity is to 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). The affinity settings for our containerized deploy-

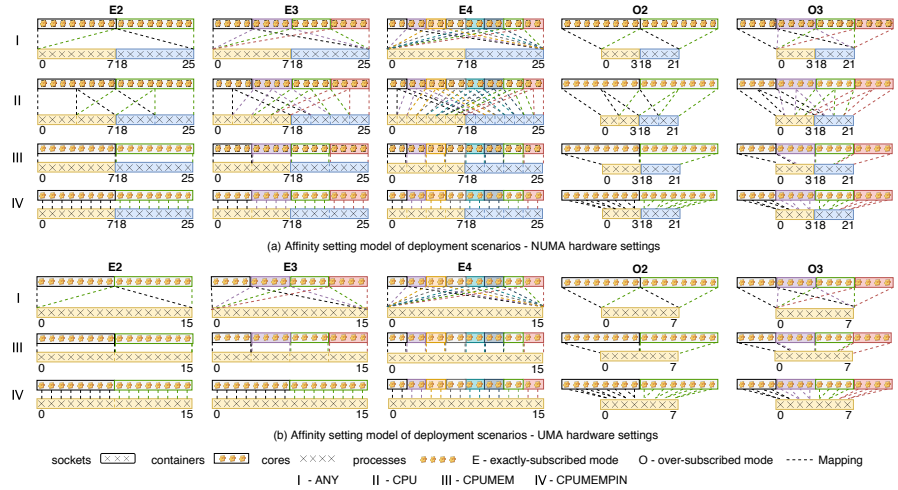

Fig. 30 Containerized deployment scenarios using affinity.

ment scenarios are shown in Figure 30. They include three different settings, namely CPU, CPUMEM, and CPUMEMPIN, which are all compared to ANY (the baseline used in the experiments in Section 4). We assume a number of containers N_{ctn}, where each one hosts a number of processes N_{mpi}, so that $N_{ctn} \times N_{mpi} = K$, which is kept constant in all the deployment scenarios (i.e., 16). For different subscription modes with ratio r, each container requests the number of cores $N_{cpu} = N_{mpi}/r$, where $r = 1$ or $r > 1$, which means the application runs on exactly-subscribed mode or over-subscribed mode, respectively. Each hardware platform setting provides a number of CPU cores and MEM nodes from one or more sockets $S = \{socket_s | s = 0, ..., N_{socket} - 1\}$, where each socket has P cores. Hence, for each application distributed in a set of containers $CTN = \{ctn_i | i = 1, ..., N_{ctn}\}$ where each one hosts a set of processes $MPI = \{mpi_j | j = 1, ..., N_{mpi}\}$, each affinity setting defines a mapping :

$$Map_{i,j} \rightarrow \begin{cases} CPU_{s,x \rightarrow y} \\ MEM_s \end{cases} \tag{1}$$

where s refers to the assigned socket and x, y refer to the range of assigned cores.

Each of the affinity settings works as follows: (I) ANY: processes do not have any processor or memory affinity, they could access all the resources provided by the hardware platform setting, and the actual distribution is decided by the operating system. Thus, the mapping of ANY scenarios could be expressed as:

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

(II) CPU: we define a specific processor affinity for each container to a set of cores from the two sockets available in the host. This can only be set in the

NUMA hardware platform setting. The mapping of CPU scenarios could be formulated as follows:

$$Map_{i,j} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{s,s\times P+(i-1)\times \frac{N_{cpu}}{N_{socket}} \rightarrow s\times P+i\times \frac{N_{cpu}}{N_{socket}}-1} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_s \end{cases} \tag{3}$$

(III) CPUMEM: we define 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:

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

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 so that each process is mapped into a specific core:

$$Map_{i,j} \rightarrow \begin{cases} CPU_{\lceil \frac{i}{N_{cps}} \rceil -1,(\lceil \frac{i}{N_{cps}} \rceil -1)\times P+((i-1)-N_{cps}\times(\lceil \frac{i}{N_{cps}} \rceil -1))\times N_{cpu}+\lceil \frac{j}{r} \rceil -1} \\ MEM_{\lceil \frac{i}{N_{cps}} \rceil -1} \end{cases} \tag{5}$$

5.3 Experimental setup

The environment, benchmarks, performance tools, statistical significance assessment methods, and container granularity settings are the same as Section 4.3. Some other settings regarding affinity are described below:

CPU affinity settings: The CPU affinity is defined by restricting the range of possible CPUs to be assigned to the containers. The `cpuset-cpus` parameter is needed for Docker to specify the set of CPUs that can be used, and for Singularity we define a cgroup.toml configuration file which sets `cpus`.

Memory affinity settings: The purpose of using memory affinity is to restrict the memory accesses of containers to the NUMA node where their assigned CPUs belong. For Docker, together with the `cpuset-cpus` parameter, the containers must be provided with the corresponding `cpuset-mems` parameter. For Singularity, we use the same strategy as Docker and specify `cpus` and `mems` options within the cgroups.toml file.

OpenMPI processes binding: Unlike the settings of ANY, CPU, and CPUMEM, where processes are free to be moved between the various CPUs

allocated to each container, CPUMEMPIN utilizes bind-to core where a more rigid procedure of ranking, mapping, and binding of processes on CPUs is carried out, actually making it a 1-to-1 process-to-processor binding. For Docker and Singularity-instance, it was necessary to configure the appropriate rankfiles that describe this behavior.

5.4 Results

Figure 31-37 show the impact when using processor and memory affinity strategies on multi-container deployments of the HPCC benchmarks on two hardware platform settings, namely the UMA and NUMA settings described before.

5.4.1 Impact of containerization technology on multi-container deployments with affinity

Figure 31-35 show the performance results of MPI communication workloads. Congruently with the results in the previous section, Singularity achieves the best performance also when using processor and memory affinity, while Docker and Singularity-instance present some performance degradation in multi-container scenarios. As discussed in the previous section, this is due to the overhead of communication through the network stack instead of using shared-memory, which depends on the amount of time spent within the MPI library and the specific MPI functions invoked. Setting affinity cannot avoid this performance degradation.

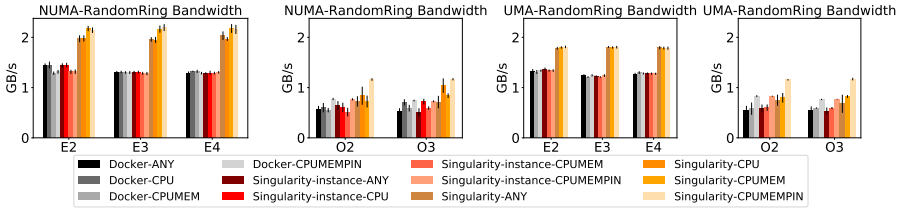

Fig. 31 Impact of affinity on RandomRing-bandwidth performance.

Figure 36-37 depict the performance results of MPI throughput benchmarks. In this case, all the containerization technologies (Docker, Singularity-instance, and Singularity) achieve the same performance if they are set with the same affinity configuration. The effectiveness of using affinity with those benchmarks is not dependent on the containerization technology because, as we discussed in the previous section, they present low inter-process communication.

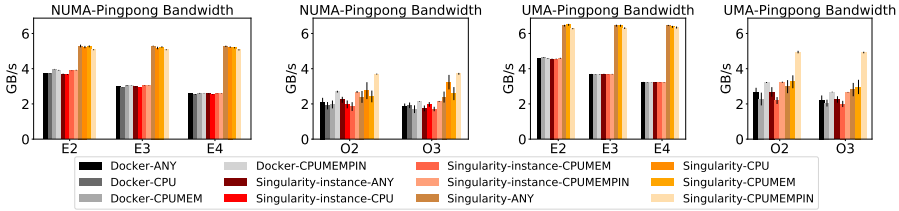

Fig. 32 Impact of affinity on Pingpong-bandwidth performance.

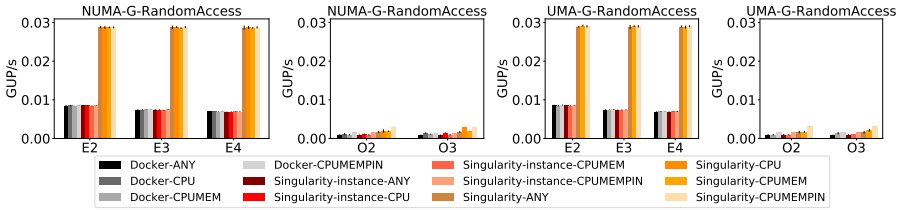

Fig. 33 Impact of affinity on G-Randomaccess performance.

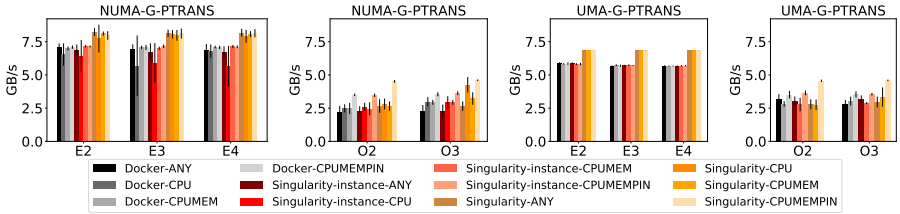

Fig. 34 Impact of affinity on G-PTRANS performance.

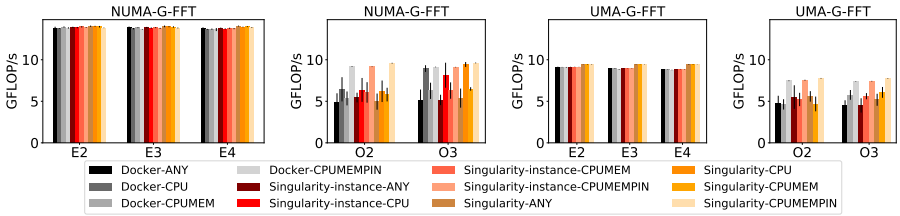

Fig. 35 Impact of affinity on G-FFT performance.

5.4.2 Impact of container granularity on multi-container deployments with affinity

As discussed previously, the impact of container granularity on multi-container deployments with affinity can be significant depending on the CPU and memory usage characteristics of each benchmark. The results in Table 6, which depict the access rate to the local memory in the NUMA setting for each benchmark on ANY scenario, show that EP-STREAM, G-PTRANS, G-FFT, and G-RandomAccess are well optimized for locality (processes mostly access the

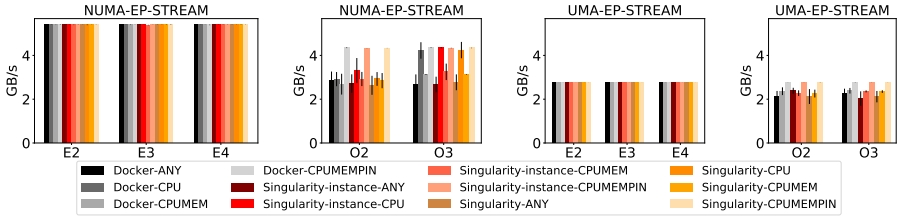

Fig. 36 Impact of affinity on EP-STREAM performance.

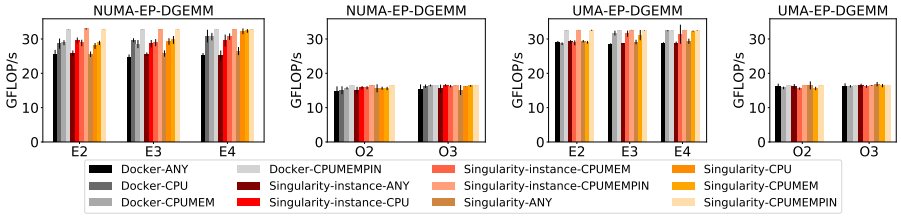

Fig. 37 Impact of affinity on EP-DGEMM performance.

Table 6 HPCC Benchmark Memory Locality Analysis.

Benchmark	Local memory access rate	
	Exact-subscribed	Over-subscribed
EP-DGEMM	56%	54%
EP-STREAM	99%	97%
G-FFT	96%	93%
G-PTRANS	98%	95%
G-RandomAccess	90%	80%
b_eff	2%	2%

local memory), while EP-DGEMM has distributed memory allocation (only 56% accesses to local memory) (b_eff performs most of its accesses to remote memory, but as it uses few memory, this is not significant for performance). Consequently, only EP-DGEMM can take advantage of using memory affinity to reduce the latency to access the memory, and the benefit of memory affinity for the other benchmarks should be negligible. Similarly, as the local memory access rates are slightly lower on over-subscribed deployment scenarios than exactly-subscribed ones, over-subscribed mode scenarios have more room for exploiting better memory affinity.

CPU and memory affinity have considerably increased the performance of EP-DGEMM in all the scenarios. Specifically, the improvement (in %) for Docker in CPU, CPUMEM, and CPUMEMPIN scenarios with respect to ANY scenarios on the NUMA setting is significant in all the exactly-subscribed scenarios (with P-values of the corresponding T-tests ranging from $9.7e^{-11}$ to $1.8e^{-4}$, clearly below 0.05): around 12%–22% (E2–E4 CPU), 13%–21% (E2–E4 CPUMEM), and 29%–33% (E2–E4 CPUMEMPIN). In the over-subscribed mode, the improvement is also significant in O2-CPUMEMX and O3 scenarios (with P-values ranging from $7.6e^{-3}$ to 0.06): 7% (O2 CPUMEM),

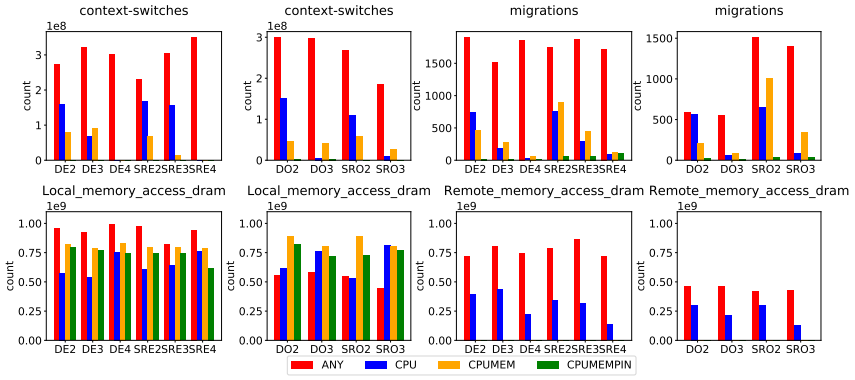

Fig. 38 Performance event counters of EP-DGEMM on Docker and Singularity for scenarios with different affinity on NUMA hardware platform setting.

11% (O2 CPUMEMPIN), 6% (O3 CPU), 7% (O3 CPUMEM), and 7% (O3 CPUMEMPIN), but not significant in O2-CPU: 2% with P-value 0.5. These performance increments are directly related with the container granularity, as finer-grained deployments provide better improvement. 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. This can be seen in the counter values in Figure 38. 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. Overall, affinity improves the cache usage and optimizes the data allocation of the EP-DGEMM application.

For EP-STREAM, G-PTRANS, G-FFT, G-RandomAccess, and b_eff benchmarks, memory affinity does not impact significantly the performance because b_eff uses few memory and the others have the memory allocated mostly in the local socket already. The impact of CPU affinity on exactly-subscribed scenarios is not significant either for those benchmarks. As shown in Figure 39, which depicts the counter values for G-FFT benchmark on the NUMA setting, the operating system can do a pretty good job to prevent unnecessary context-switches on exactly-subscribed scenarios.

On the other side, CPU affinity can increase the performance in over-subscribed scenarios for those benchmarks. This is especially noticeable with CPUMEMPIN affinity configuration (e.g., Docker shows significant improvements from 28% to 87% in scenario O2, with P-values clearly lower than 0.05 ranging from $6.7e^{-7}$ to $1.8e^{-4}$, and from 17% to 80% in scenario O3, with P-values also lower than 0.05 ranging from $8.3e^{-6}$ to $1.8e^{-4}$), and also with CPU configuration in scenario O3 (e.g., improvements ranging from 31% to 77% are significant for all the benchmarks but b_eff(PingPong), with P-values ranging from $7.6e^{-4}$ to $1.9e^{-3}$). ANY configuration is also generally worse than CPU configuration in scenario O2 for all the benchmarks but b_eff(PingPong) (with

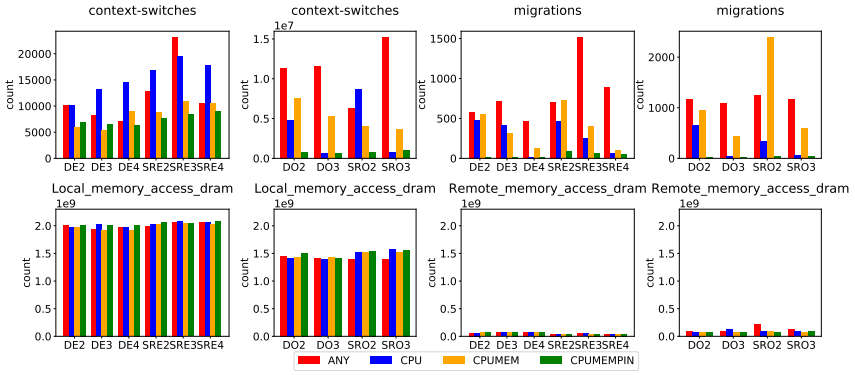

Fig. 39 Performance event counters of G-FFT on Docker and Singularity for scenarios with different affinity on NUMA hardware platform setting.

improvements from 2% to 31%, but most of them not statistically significant as the P-values are higher than 0.05) and CPUMEM configuration in scenario O3 (with improvements from 11% to 29%, which are halfway significant with P-values mostly ranging from 0.001 to 0.3). Results for CPUMEM in O2 are inconclusive, as all P-values of the T-tests are higher than 0.05, ranging from 0.2 to 0.79. As shown in Figure 39, in over-subscribed scenarios, CPUMEMPIN and CPU configurations have less cpu-migrations and context-switches than CPUMEM, which has also less than ANY. Processes in O3-CPU are using cores belonging to two sockets. As migrations between sockets are more expensive (e.g. expensive computation for iterating all the runqueues, expensive cache misses, and synchronization), the scheduler tries more to avoid them [21], something that does not happen in O3-CPUMEM, where the used cores belong to the same socket. Similarly, the improvement with CPU and CPUMEM in scenario O3 is also higher than in scenario O2, because O3 allows to use only one core per socket, which is effectively encouraging 1-to-1 process-to-processor pinning.

5.4.3 Impact of the cgroup scheduling on multi-container deployments with affinity

In section 4.4.2, we assessed the impact of the cgroup scheduling performed by CFS on ANY scenarios. CFS tried to maintain fair time allocation among cgroups, but incurred some performance degradation on over-subscribed mode scenarios due to load imbalance among the various processors. In this section, we assess the impact of the cgroup scheduling on multi-container deployments with affinity, to check if affinity could help to overcome this degradation.

Figure 40 shows the EP-DGEMM performance on CPU and CPUMEM affinity scenarios with different number of containers. All the scenarios provide the same performance and do not incur performance degradation, even in containerization technologies which create a different cgroup per container (e.g.

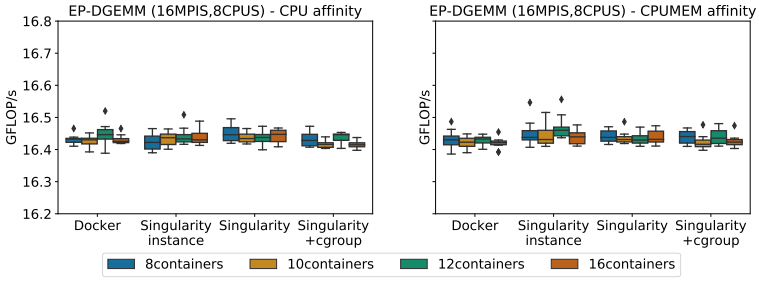

Fig. 40 Performance comparison of EP-DGEMM on CPU/CPUMEM scenarios with different number of containers.

Docker, Singularity-instance, Singularity+cgroup). This can be confirmed by means of T-tests for deployments with more than 8 containers regarding the 8-containers deployment, which have P-values ranging from 0.06 to 0.97, all higher than 0.05 and hence showing no significant difference. CPU affinity is able to overcome the CFS load imbalance problem because processes are deployed explicitly in fixed processors, which avoids load balancing by the scheduler.

5.4.4 Impact of the hardware platform setting on multi-container deployments with affinity

As discussed in previous sections, the NUMA and UMA hardware platform settings can provide different performance for specific benchmarks depending on their characteristics. This happens also with multi-container deployments with affinity. A significant difference is that the UMA setting can only take advantage of CPU affinity not memory affinity, since all the memory accesses on the UMA setting are already local. Regarding CPU affinity, its performance impact in the UMA setting follows the same trend we discussed before for the NUMA setting, being clearly visible in over-subscribed scenarios for some benchmarks, where CPUMEM and CPUMEMPIN configurations on UMA are better than ANY, because they reduce the number of cpu-migrations and context-switches. In particular, EP-STREAM, G-PTRANS, G-FFT, G-RandomAccess, and b_eff benchmarks show significant performance improvements ranging from 11% to 87% for O2-CPUMEMPIN in Docker, with P-values of the T-tests ranging from $3.4e^{-6}$ to 0.04, and from 20% to 101% for O3-CPUMEMPIN, with P-values ranging from $1.1e^{-7}$ to $4.7e^{-4}$. EP-STREAM, G-FFT, and G-RandomAccess benchmarks also show significant performance improvements ranging from 6% to 64% for O3-CPUMEM, with P-values ranging from $1.8e^{-4}$ to 0.037. The results of those benchmarks for O2-CPUMEM are inconclusive, as the performance differences are small (from -3% to 11%) and generally not statistically significant (with P-values ranging from 0.04 to 0.73).

5.4.5 Summary

The findings from our previous evaluation of the impact of processor and memory affinity on multi-container deployments are as follows:

- Multi-container deployments with affinity cannot prevent the performance degradation of Docker and Singularity-instance with MPI communication workloads due to the execution of containers on separated network namespaces. With MPI throughput workloads, all the containerization technologies achieve the same performance if they are set with the same affinity configuration.
- As containers do not virtualize memory, partitioning HPC workloads into multiple containers does not show benefits from a memory translation perspective, but finer-grained container granularity can improve the performance on multi-container deployments with affinity depending on the CPU and memory usage characteristics of each benchmark. Memory affinity reduces the number of accesses to the remote memory in benchmarks with distributed allocated memory, while CPU affinity restricts the cores that processes can be allocated, which reduces the number of cpu-migrations and context switches, especially in over-subscribed scenarios.
- 1-to-1 process-processor pinning scenarios (i.e. CPUMEMPIN scenarios) provide the best performance, but less strict affinity configurations can be acceptable alternatives when 1-to-1 pinning is not straight-forward (e.g., in over-subscribed scenarios where the number of processes is not a multiple of the number of processors).
- On over-subscribed mode, CPU affinity is able to overcome the CFS load imbalance problem causing performance degradation, because processes are deployed explicitly in fixed processors and this eliminates the need to balance load by the scheduler.
- Memory affinity does not provide added benefits in the UMA hardware platform setting, since memory accesses are already local. CPU affinity improves the performance of some benchmarks in over-subscribed scenarios (as in the NUMA setting), by reducing the number of cpu-migrations and context-switches.

6 Conclusion and future work

This paper presented a performance comparison of multi-container deployment schemes for HPC workloads. In order to understand the performance impact of different deployment scenarios, we selected HPCC workloads that exhibit different communication patterns, memory accesses, and computation. We executed the various deployment schemes on NUMA and UMA hardware platform settings with different subscription modes (exactly-subscribed and over-subscribed). Our research revolved around the above settings to understand the performance of different containerization technologies (e.g. Docker and Singularity), especially in terms of the impact of granularity of containers

and the effectiveness of using processor and memory affinity for the various deployment schemes.

We concluded that some trade-offs need to be taken into account when choosing multi-container deployment schemes for HPC workloads. Docker and Singularity-instance incur some performance degradation for MPI communication workloads running on multiple containers, because the processes running on separated containers are deployed on isolated network namespaces. Multi-container deployments with affinity cannot prevent this performance degradation, but the degradation could be avoided by enabling shared-memory among the distinct containers and making the MPI engine aware of that shared-memory area. Singularity, which can use shared-memory for communication, is not affected by this issue.

Workloads with low amount of inter-process communication do not incur performance degradation with any containerization technology and can benefit from finer-grained deployments because they simplify the scheduling in a similar way to when processes are pinned explicitly. Finer-grained container granularity can improve also the performance on multi-container deployments with affinity depending on the CPU and memory usage characteristics of each benchmark, especially in over-subscribed scenarios. 1-to-1 process-processor pinning provides the best performance, but less strict affinity configurations can be acceptable alternatives when 1-to-1 pinning is not straight-forward.

On over-subscribed mode, some performance degradation is due to the scheduling of cgroups by Linux CFS, which results in an imbalanced allocation of processes to processors. CPU affinity allows to overcome this problem, because processes are deployed explicitly in fixed processors and this eliminates the need to balance load by the scheduler.

The performance difference between the hardware platform settings is not directly related with any containerization technology or container granularity, but with the application and hardware setting characteristics, such as the cache usage or the memory bandwidth. Memory affinity does not provide added benefits in the UMA setting but improves the performance of benchmarks with distributed memory allocation in the NUMA setting. CPU affinity improves the performance of some benchmarks in over-subscribed scenarios on both hardware platform settings, by reducing the number of cpu-migrations and context-switches.

In the future, we plan to evaluate our multi-container deployment schemes on multiple hosts that communicate through different network fabrics and protocols (e.g., TCP/IP on Ethernet, TCP/IP on Infiniband (IPoIB), and RDMA on Infiniband). Also, we will use insights about the performance of multi-container deployments, especially those regarding the impact of the container granularity and the CPU and memory affinity, to derive placement policies when deploying HPC workloads which can get better utilization of the resources while maintaining application performance [26].

Acknowledgements We thank Lenovo for providing the technical infrastructure 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 2020 FI-B 00257.

References

1. Alam, S., Barrett, R., Bast, M., Fahey, M.R., Kuehn, J., McCurdy, C., Rogers, J., Roth, P., Sankaran, R., Vetter, J.S., et al.: Early evaluation of IBM BlueGene/P. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08), pp. 1–12. IEEE (2008). DOI 10.1109/SC.2008.5214725
2. Arango, C., Dernat, R., Sanabria, J.: Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. CoRR **abs/1709.10140** (2017)
3. Azab, A.: Enabling Docker Containers for High-Performance and Many-Task Computing. In: Proceedings of the 2017 IEEE International Conference on Cloud Engineering (IC2E), pp. 279–285 (2017). DOI 10.1109/IC2E.2017.52
4. Bacik, J.: Cpu scheduler imbalance with cgroups. URL `https://josefbacik.github.io/kernel/scheduler/cgroup/2017/07/24/scheduler-imbalance.html`
5. Banerjee, A., Mehta, R., Shen, Z.: NUMA Aware I/O in Virtualized Systems. In: Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 10–17 (2015). DOI 10.1109/HOTI.2015.17
6. Bermejo, B., Juiz, C.: On the classification and quantification of server consolidation overheads. Journal of Supercomputing pp. 1–21 (2020). DOI 10.1007/s11227-020-03258-2
7. Cheng, Y., Chen, W., Chen, X., Xu, B., Zhang, S.: A user-level numa-aware scheduler for optimizing virtual machine performance. In: Revised Selected Papers of the 10th International Symposium on Advanced Parallel Processing Technologies - Volume 8299, APPT 2013, pp. 32–46. Springer-Verlag, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-45293-2_3
8. Chung, M.T., Quang-Hung, N., Nguyen, M., Thoai, N.: Using Docker in High Performance Computing applications. In: Proceedings of the 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), pp. 52–57 (2016). DOI 10.1109/CCE.2016.7562612
9. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172. IEEE (2015). DOI 10.1109/ISPASS.2015.7095802
10. Google: Cgroups-cpus. URL `https://kernel.googlesource.com/pub/scm/linux/kernel/git/glommer/memcg/+/cpu_stat/Documentation/cgroups/cpu.txt`
11. Halácsy, G., Ádám Mann, Z.: Optimal energy-efficient placement of virtual machines with divisible sizes. Information Processing Letters **138**, 51–56 (2018). DOI 10.1016/j.ipl.2018.06.003
12. HPC advisor council: HPCC Performance Benchmark and Profiling (2015). URL `https://hpcadvisorycouncil.com/pdf/HPCC_Analysis_and_Profiling_Intel_E5-2697v3.pdf`
13. HPC wire: Sylabs releases singularity 3.0 container platform; Cites AI Support (2018). URL `https://www.hpcwire.com/2018/10/08/sylabs-releases-singularity-3-0-container-platform-cites-ai-support/`
14. Ibrahim, K.Z., Hofmeyr, S., Iancu, C.: Characterizing the performance of parallel applications on multi-socket virtual machines. In: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 1–12. IEEE (2011). DOI 10.1109/CCGrid.2011.50
15. Ibrahim, K.Z., Hofmeyr, S., Iancu, C.: The Case for Partitioning Virtual Machines on Multicore Architectures. IEEE Transactions on Parallel and Distributed Systems **25**(10), 2683–2696 (2014). DOI 10.1109/TPDS.2013.242
16. 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 Transactions on Parallel and Distributed Systems **22**(6), 931–945 (2011). DOI 10.1109/TPDS.2011.66
17. Jha, D.N., Garg, S., Jayaraman, P.P., Buyya, R., Li, Z., Morgan, G., Ranjan, R.: A study on the evaluation of HPC microservices in containerized environment. Concurrency and Computation (March), 1–18 (2019). DOI 10.1002/cpe.5323
18. Jha, D.N., Garg, S., Jayaraman, P.P., Buyya, R., Li, Z., Ranjan, R.: A Holistic Evaluation of Docker Containers for Interfering Microservices. In: Proceedings of the 2018 IEEE International Conference on Services Computing (SCC), pp. 33–40 (2018). DOI 10.1109/SCC.2018.00012
19. Kuity, A., Peddoju, S.K.: Performance Evaluation of Container-Based High Performance Computing Ecosystem Using OpenPOWER. In: J.M. Kunkel, R. Yokota, M. Taufer, J. Shalf (eds.) High Performance Computing, ISC High Performance 2017, Lecture Notes in Computer Science, vol. 10524, pp. 290–308. Springer International Publishing, Cham (2017). DOI 10.1007/978-3-319-67630-2_22
20. Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: Scientific containers for mobility of compute. PLOS ONE **12**(5), e0177459 (2017). DOI 10.1371/journal.pone.0177459
21. Lozi, J.P., Lepers, B., Funston, J., Gaud, F., Quéma, V., Fedorova, A.: The Linux Scheduler: A Decade of Wasted Cores. In: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys'16. Association for Computing Machinery (2016). DOI 10.1145/2901318.2901326
22. 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 (SC'06) (2006). DOI 10.1145/1188455.1188677
23. Luszczek, P. and Koester, D.: HPC Challenge v1.x Benchmark Suite. SC'05 Tutorial, Seattle, Washington (2005). URL `http://icl.cs.utk.edu/news_pub/submissions/HPCChallengeTutorialDPKPL22Nov2005.pdf`
24. Maliszewski, A.M., Griebler, D., Schepke, C., Ditter, A., Fey, D., Fernandes, L.G.: The NAS Benchmark Kernels for Single and Multi-Tenant Cloud Instances with LXC/KVM. In: Proceedings of the 2018 International Conference on High Performance Computing Simulation (HPCS), pp. 359–366 (2018). DOI 10.1109/HPCS.2018.00066
25. Mann, H.B., Whitney, D.R.: On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. Ann. Math. Statist. **18**(1), 50–60 (1947). DOI 10.1214/aoms/1177730491
26. Menouer, T.: KCSS: Kubernetes container scheduling strategy. Journal of Supercomputing pp. 1–27 (2020). DOI 10.1007/s11227-020-03427-3
27. OpenMPI Team: Can i force aggressive or degraded performance modes? URL `https://www.open-mpi.org/faq/?category=running`
28. OpenMPI Team: Can I oversubscribe nodes (run more processes than processors)? URL `https://www.open-mpi.org/faq/?category=running`
29. Perarnau, S., Essen, B.C.V., Gioiosa, R., Iskra, K., Gokhale, M.B., Yoshii, K., Beckman, P.: Argo. Operating Systems for Supercomputers and High Performance Computing (2019). DOI 10.1007/978-981-13-6624-6_12
30. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A Tool to Visualize and Analyze Parallel Code. In: Proceedings of the 18th World Occam and Transputer User Group Technical Meeting, pp. 9–13. IOS Press (1995)
31. Rao, J., Wang, K., Zhou, X., Xu, C.: Optimizing virtual machine scheduling in NUMA multicore systems. In: Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 306–317 (2013). DOI 10.1109/HPCA.2013.6522328
32. Roloff, E., Diener, M., Carissimi, A., Navaux, P.O.A.: High Performance Computing in the cloud: Deployment, performance and cost efficiency. In: Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science, pp. 371–378 (2012). DOI 10.1109/CloudCom.2012.6427549
33. 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). DOI 10.1109/IPDPS.2019.00066

34. Saha, P., Beltre, A., Govindaraju, M.: Scylla: A mesos framework for container based MPI jobs. CoRR **abs/1905.08386** (2019)
35. Saha, P., Beltre, A., Uminski, P., Govindaraju, M.: Evaluation of Docker Containers for Scientific Workloads in the Cloud. In: Proceedings of the Practice and Experience on Advanced Research Computing, PEARC'18. Association for Computing Machinery (2018). DOI 10.1145/3219104.3229280
36. Sande Veiga, V., Simon, M., Azab, A., Fernandez, C., Muscianisi, G., Fiameni, G., Marocchi, S.: Evaluation and Benchmarking of Singularity MPI containers on EU Research e-Infrastructure. In: Proceedings of the 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp. 1–10 (2019). DOI 10.1109/CANOPIE-HPC49598.2019.00006
37. Shapiro, S.S., Wilk, M.B.: An analysis of variance test for normality (complete samples). Biometrika **52**(3-4), 591–611 (1965). DOI 10.1093/biomet/52.3-4.591
38. Sharma, P., Chaufournier, L., Shenoy, P., Tay, Y.C.: Containers and Virtual Machines at Scale. In: Proceedings of the 17th International Conference on Middleware, pp. 1–13 (2016). DOI 10.1145/2988336.2988337
39. Sterling, T., Anderson, M., Brodowicz, M.: The Essential Resource Management. In: High Performance Computing, Chapter 5, pp. 141–190. Morgan Kaufmann, Boston (2018). DOI 10.1016/B978-0-12-420158-3.00005-8
40. Tesfatsion, S.K., Klein, C., Tordsson, J.: Virtualization Techniques Compared: Performance, Resource, and Power Usage Overheads in Clouds. In: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, pp. 145–156. Association for Computing Machinery (2018). DOI 10.1145/3184407.3184414
41. Torrez, A., Randles, T., Priedhorsky, R.: HPC Container Runtimes have Minimal or No Performance Impact. In: Proceedings of the 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp. 37–42 (2019). DOI 10.1109/CANOPIE-HPC49598.2019.00010
42. 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). DOI 10.1109/IPDPS.2011.68
43. Vmware: Virtualizing High-Performance Computing (HPC) Environments: Reference Architecture (September) (2018)
44. Wang, Y., Evans, R.T., Huang, L.: Performant Container Support for HPC Applications. In: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning), PEARC'19, pp. 1–6. Association for Computing Machinery (2019). DOI 10.1145/3332186.3332226
45. Welch, B.L.: The Generalization of Student's Problem When Several Different Population Variances Are Involved. Biometrika **34**(1-2), 28–35 (1947). DOI 10.1093/biomet/34.1-2.28
46. Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A.F.: Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In: Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 233–240 (2013). DOI 10.1109/PDP.2013.41
47. Xing, F., You, H., Lu, C.: HPC benchmark assessment with statistical analysis. Procedia Computer Science **29**, 210–219 (2014). DOI 10.1016/j.procs.2014.05.019
48. Yang, S., Wang, X., An, L., Zhang, G.: Yun: A High-Performance Container Management Service Based on OpenStack. In: Proceedings of the 2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC), pp. 202–209 (2019). DOI 10.1109/DSC.2019.00038
49. 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). DOI 10.1109/CloudCom.2017.40