Kuity and Peddoju
Journal of Cloud Computing (2023) 12:178
https://doi.org/10.1186/s13677-023-00546-z

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Investigating performance metrics for container-based HPC environments using x86 and OpenPOWER systems

Animesh Kuity[1*] and Sateesh K. Peddoju[1]

Abstract

Container-based High-Performance Computing (HPC) is changing the way computation is performed and reproduced without sacrificing the raw performance compared to hypervisor-assisted virtualization technologies. It primarily supports continuously evolving data-intensive applications such as computational fluid dynamics, seismic tomography, molecular biology, and Proteomics. OpenPOWER systems, unlike the x86 systems, use the POWER-compliant processor to exploit instruction-level and thread-level parallelism heavily. In our previous work, we designed and developed a Containerized HPC environment (cHPCe) from the scratch using Linux namespaces on OpenPOWER systems. This paper aims to provide an in-depth performance analysis of the Containerized HPC environment using x86 systems and Containerized HPC environment using the OpenPOWER system, on systems' subcomponents, processor, memory, interconnect, and IO. This sub-component analysis provides an insight on several aspects of the system performance. To the best of our knowledge, no research has been reported yet for such a comparative analysis that designs cHPCe for both x86 and OpenPOWER systems. The performance of the developed cHPCe is compared with BareMetals, and VMs using the benchmarks HPCC, and IOZone. Our experimental results achieve 0.13% less compute performance penalty at its peak performance on cHPCe compared to the BareMetal-based solution for x86 systems. In contrast, a VM-based solution introduces an overhead of 20% and 4.83% in x86 and OpenPOWER cases, respectively. Moreover, the x86 and OpenPOWER systems observe inconsistent behavior for memory performance with a worst-case penalty of 9.68% and 6.64% compared to achieved peak performance, respectively. However, similar behavior is reported for cHPCe with an overhead of less than 3% and 2% in the worst case for the latency and bandwidth, respectively, compared to the BareMetal for network and disk performance. Our experimental results reveal that the containerized OpenPOWER environment represents a viable alternative to the counterpart x86 environment for the HPC solution.

Keywords Cloud computing, HPC, Virtual machine, Performance evaluation, Container technology, HPCC, OpenPOWER system

Introduction

The emerging High-Performance Computing (HPC) solutions target simplifying the implementation of complex HPC environments. The primary goal of the HPC solutions is to maximize the productivity and efficiency without compromising on the performance, especially to support dynamic data-intensive workloads. On the other hand, users demand cloud-like HPC environments with the comparable performance of BareMetal systems.

*Correspondence:
Animesh Kuity
akuity@cs.iitr.ac.in
[1] High Performance Computing Lab, Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, Haridwar, Roorkee, India

Springer Open

© The Author(s) 2023. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

The HPC users desire the support of the complex software stacks specific to their applications' needs. It is often challenging to install, port, and manage them due to their critical dependency on the administrator. The Cloud service providers (CSP) in the past used hypervisor supported VM-based solutions to satisfy the need for flexible environments. But the usefulness of such VM-based cloud in HPC remains an open research challenge due to its inherent overhead in almost all aspects, especially memory, I/O, and interconnect spaces [1–3]. In this direction, the recently skyrocketing Linux container technology, a lightweight process virtualization mechanism, has drawn the attention of the HPC community because of its near-native performance. The Linux container offers an isolated and portable environment with all the runtime needs that an application requires to execute. It provides an abstract environment with the help of the Linux kernel features, namely, *namespaces*[1] and *cgroups*[2]. The *namespaces* provide the abstraction of the system resources for processes, and *cgroups* handle the resource usage. In this work, we develop a container-based HPC environment (cHPCe) for both x86 and OpenPOWER systems and analyze the performance.

Motivation The primary challenge of the HPC solution providers is to provide a dynamic HPC environment for scientific and technical computing users so that the focus would be on computing rather than underlying background details. Hence, the HPC solution architects aim to provide the following:

1. *Maximize throughput of the HPC environment.*
2. *Minimize environment management effort.*
3. *Alleviate on the problem of research reproducibility.*

The performance analysis is one of the critical approaches to observe the bottlenecks and maximize the throughput of the provided environment. Although few authors, such as [4–9] assessed and analyzed the performance effect of container-based HPC solutions, however, there are several research gaps as discussed below:

(a) The lack of research that reports the development of the container-based HPC environments on OpenPOWER machines along with its performance analysis against x86 systems.
(b) None of the research works has examined the performance of BareMetal, container, and VM for all the HPC sub-components, i.e., CPU, Memory, I/O, Interconnect, and disk on both the systems.
(c) Most of the works on container employed either on Docker or OpenVZ or Linux Vserver or LXC on a virtual machine. These environments do not support the workload manager plug-in integration readily for HPC.

Hence, the existing solutions fail to give us an in-depth insight into the usability of containers and their impact on HPC environments. To take the advantage of the system potential, it is necessary to understand and carefully examine the kernel execution platforms and their subsystems keeping algorithmic needs. They may depend on specific characteristics of the individual application, like, whether compute-intensive or memory-intensive, or user-specific domain, and the hardware on which these models are executed. These studies make it possible to correctly map the application kernel to the heterogeneous hardware, and also to achieve maximum performance on a particular HPC system and its containerized dependencies. Moreover, advances in container technology in the cloud necessitate a continual evaluation of the suitability of HPC for a variety of applications.

Contribution The proposed work explores container-based HPC environments that rely on namespace using a) OpenPOWER, and b) x86 systems, to support research reproducibility and deal with software dependencies without disrupting the existing environment. We use HPCC, and IOZone benchmarks to compare the performance of the container-based HPC environment with BareMetals and VMs. Further, the experiments assess the performance by varying the number of threads to exploit the simultaneous multithreading capability of the systems fully. The main research contributions of this work are as follows:

- *container-based High-Performance Computing environments (cHPCes) that rely on namespace using OpenPOWER and x86 systems are presented.*
- *The performance of cHPCes is evaluated exhaustively. The study mostly motivates the performance analysis of the MPI-style cluster rather than a single instance.*
- *The performance of the proposed cHPCes is compared with the similar environments made of VMs and BareMetal servers.*

This work extends our previous contribution [10] with an exhaustive comparative performance analysis of BareMetal, container, and VM-based environment on the x86 and OpenPOWER systems with the architecture-specific compiler and the libraries. Our work presents a comprehensive evaluation and analysis of HPC environments that contain x86 and OpenPOWER systems to systematically understand each subsystem's maximum achievable

[1] https://lwn.net/Articles/531114/
[2] https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

performance, restrictions, and bottlenecks to be considered in future design. Moreover, our findings reveal the suitability of Container for HPC, and most importantly, it clarifies the perspective of using OpenPOWER or x86 systems at the hardware level for different categories of applications depending on the potential requirement of various subcomponents of the underlying physical system.

Organization The rest of the paper is organized as follows: "Background" section presents a detailed overview of the container by comparing it with a traditional hypervisor-based solution. "Related work" section discusses the current related work. "Container-based HPC environment" section describes the proposed prototype model on the Container-based HPC environment (cHPCe). We present the experimental testbed in "Experimental testbed" section. "Performance evaluation on HPC environments" section presents the detailed performance evaluations and the experimental results of different benchmark applications on environments made of containers, BareMetals, and VMs. "Conclusion & future work" section concludes the work and outlines future work.

Background

HPC is a highly mature technology treated as the domain of experts using the overpriced proprietary systems. The requirement of computational scientists for HPC environment is detailed in [11]. Their primary focus is on timely executing and maximizing science performed on the computational workload with allocated resources by optimizing the performance benefit. This approach mainly aims for superior portability, reproducibility, manageability, productibility, and availability, apart from maximizing the utilization. The important terms like Virtual Machine (VM), container, software dependency problem, MPI cluster, and OpenPOWER system are introduced below.

Virtual machine: The virtual machine monitor (VMM) or hypervisor allows us to run multiple guest operating systems that share the same underlying hardware resources. It virtualizes all the physical resources of the host to provide an environment to the users by interacting with the host OS or hardware. The commonly used hypervisor solutions include KVM, VMWare, Oracle VM VirtualBox, Microsoft's Hyper-V, etc. The hypervisor solution, KVM, is used as the representative solution for our experimental setup. KVM provides a full virtualization solution for Linux on OpenPOWER and x86 systems. We use QEMU integrated KVM solution to deploy the VM. QEMU delivers emulated hardware to achieve near-native speed.

Container: The Linux container technology, a lightweight process virtualization mechanism, allows executing multiple isolated user-space instances that share the same kernel as the host OS. The container provides a secluded runtime environment but can share up to libraries or binaries with the host OS (for instance, an application container). The Linux container relies on the two mature kernel features, namely, *namespaces* and *cgroups*. The *namespace* feature provides an isolated view of the global system resources to each process. Various kinds of namespace, such as Mount, UTS, IPC, Net, PID, and User, offer abstraction for the global system resources to a group of processes. *Cgroup* feature restricts how much resources a group of processes can use. Container presents little or no overhead to the system resources compared to BareMetal. Several container-based solutions are available nowadays, such as LXC, Docker, OpenVZ, Solaris Container, etc. Our experimental setup utilizes the *namespace* features to deploy raw containers.

Software dependency problem: HPC users want the execution of the applications in the same environment that they utilize for the development purpose. Such a dependency adds more challenges to the system administrators. The system administrators need to deal with a suitable environment for rapid development of applications including incredibly data-intensive applications, libraries, and tools demanded by the specific scientific community. It is cumbersome to provide an environment that changes over time as libraries and compilers get updated very frequently with the change in HPC systems. On the other hand, the user may be interested in having application-specific tools that are sometimes difficult to install and port along with all the dependencies. This dependency hell may arise due to the varying communities. For example, the genomics and biologist, mostly, demand Ubuntu as their base image with a specific version of Python and/or Perl. In contrast, High-energy physicists may ask for Scientific Linux with specific compilers, libraries, and scripting tools. HPC users desire to have a cloud-like flexible environment, where they can quickly move their desktop environment using User Defined Images (UDI) capability with the potential to have BareMetal performance without dealing with nasty underlying environmental details. Linux container technology has emerged in a revolutionary way to satisfy user demands for a flexible computing environment. The container can provide a whole stack of runtime environments similar to a home development environment. In order to accelerate the research reproducibility, the containers are used to resolve dependencies and portability issues.

Consequently, users can comfortably bring their images used in their development environment to perform their experiments in supercomputing like data centers.

MPI cluster: MPI is a de-facto standard programming paradigm for distributed parallel application processes to communicate across multiple nodes in the HPC environment. OpenMPI, MPICH, and MVAPICH2 are popular examples of MPI implementations. MPI communication can happen in different forms, such as collective, point-point, or one-sided. It offers high-level primitives to the application developers to hide the data movement complexity in the NUMA node through various interconnects under different configurations. Advanced MPI libraries implementation maintains several performance-aware knobs to get the most out of it. CUDA-aware MPI libraries can directly handle GPU-resident data by explicitly managing memory in MPI primitives, improving performance and productivity [12]. Most previous works present the performance analysis of containerized environments either on a single standalone system or deployed containers inside the virtualized environment. Their models fail to provide an in-depth understanding of overheads imposed due to distributed memory compute systems. Our experimental results show how system-level advancement improves inter-process and intra-process communication for heterogeneous systems computation.

OpenPOWER system: OpenPOWER Foundation, backed by IBM's POWER processor [13], accelerates the adaptation of an open server architecture to optimize and innovate various advanced hardware technologies and software co-designs for data centers to implement various scientific applications. Each POWER8 processor carries up to 12 cores per socket, with each core holding eight hardware threads, 64KB L1 data cache, and 32KB L1 instruction cache. Additionally, each core supports a 512KB L2 cache and 8MB L3 cache, along with accompanying up to 128MB off-chip L4 eDRAM. Further, the on-chip memory controller can support 1TB RAM and 230GB/s sustainable memory bandwidth, and 48GB/s I/O to the other part of the system. The processor clock operates at rates between 2.5 to 5GHz. POWER8 processor incorporates enhanced prefetching features like data prefetch depth awareness, instruction speculation awareness, and Coherence Attach Processor Interface (CAPI), providing a direct communication link between the POWER8 CPU and co-processor and peripherals.

Related work

The significant contribution in this field mainly falls into two categories: x86-backed solution and OpenPOWER-backed solution.

x86-backed solution. David et al. [5] observed the unexploited potential of HPC in the cloud and conducted several benchmark experiments to learn the feasibility of a container-based HPC environment that shows its near-native performance. This performance metric compares CPU, network, and inter-process communication against KVM. This work utilizes the virtualized environment of VM for their experimental setup. Similarly, Felter et al. [6] took into account the CPU performance metric and presented the performance comparison of KVM with Docker container. The server application MySQL is used to conclude that both the techniques require tuning related to I/O operation.

Migual et al. [7] carried out an extensive performance investigation for container-based virtualization such as LXC, OpenVZ, and LinuxVserver for HPC compared to Xen. The performance overhead includes Compute, Memory, Disk, Network, and isolation [14] features of the container compared to the virtual machine to conclude the applicability of the container technology in HPC. The authors claim that LXC gives the higher performance in most of the cases, except in isolation, where hypervisor-based solution dominates.

Scheepers et al. [15] also encouraged the above observation based on performance comparison using an application and inter-VM communication microbenchmark. Single Root I/O Virtualization (SR-IOV) significantly mitigates the high I/O overhead problem that usually occurs in I/O intensive HPC workloads. A solution to the communication overhead is also addressed in earlier work [16] using a Software-Defined Artificial Neural Network (SD-ANN) switch based on a Fiber channel with a predicted neural network technique. Using intelligent congestion control and QoS implementation, the model offers a high throughput and low latency network for lossless data transmission. Their approach is suitable for reducing internode network latency. Though container-based solutions outshine VM-based solutions [17] regarding inter-process communication, network latency, and bandwidth, however, hypervisor-based VM presents a more robust isolation feature. To enhance containers' isolation, Mavridis et al. [18] utilized the combination of VMs and containers, and also quantified the performance overhead experimentally on various virtual machines. Similarly, Li et al. [9] presented the performance difference between the stand-alone Docker container with a stand-alone virtual machine using several benchmarks. The authors revealed that the performance difference could arise not only due to a feature-by-feature basis but also a job-by-job basis.

To the best of our knowledge, the existing state-of-the-art solutions fail to provide an in-depth analysis of the performance results by examining all the HPC

Table 1 Summary of related work w.r.t system used- and sub-components stressed on BareMetal Vs. VM Vs. container

	x86 Only	OpenPOWER Only	Both
David et al. [5]	C,B,L	x	x
Felter et al. [6]	I	x	x
Li et al. [9]	C,M,B,I	x	x
Zhang et al. [17]	B,L	x	x
Kuity et al. [10]	x	C,M,B,L,I	x

C, M, B, L and I denotes Compute, Memory, Bandwidth, Latency, and I/O respectively

subcomponents (comprising CPU, Memory, I/O, and Interconnect) considering all BareMetal, container, and VM altogether in their assessments. The proposed solutions are unable to provide a glimpse of the scope of improvement in each HPC subcomponent. Several contributions on container employed either Docker or OpenVZ or Linux Vserver or LXC on a virtual machine. These environments do not support the workload manager plug-in integration readily for HPC.

OpenPOWER-backed solution. Adinetz et al. [19] analyzed the performance of the POWER8-based system using STREAM and OpenMP micro-benchmark suite. The authors have also provided insight into how the POWER8-based system can be exploited to efficiently utilize several applications such as LBM, MAFIA, and NEST. Lu et al. [20] extensively evaluated the performance of OpenPOWER systems for processing big-data workload using an improved RDMA-based Hadoop RPC engine. They also discussed the architecture-aware tuning and CPU-affinity policies to enhance the performance of RDMA-based communication. Similarly, Reguly et al. [21] reported the performance of the several application benchmarks such as Black-Scholes computations, Rolls-Royce Hydra, and CFD applications on the OpenPOWER system.

In our previous work [10], the performance comparison of container-based HPC to the BareMetal and VM-based environment on the OpenPOWER system is presented. The initial implementation of the environment is evaluated, and the preliminary results were reported based on the experimental setup without architecture-specific libraries. Therefore, our experiments could provide only limited insight into our performance analysis. Moreover, the proposed work in this paper presents a comparative analysis of our proposed environments on OpenPOWER and x86 systems, and most importantly, this work is extensively different from our previous article in terms of rigorous findings, and exhaustive analysis of those findings on the x86 and OpenPOWER systems.

Table 1 presents the comparison of the existing contributions based on the different types of environments used for analysis. Table 2 summarizes the existing contributions in the container space for the HPC environment. However, to the best of our knowledge, no work is reported on container-based HPC environments developed on OpenPOWER machines that show the performance comparison with its x86 counterpart.

Discussions. To fully utilize the compute system efficiency, it is essential to carefully identify the kernel execution hardware based on the algorithmic needs. These requirements rely on the model of the individual application characteristics, namely compute-bound or memory-bound, or user-specific domain, and the hardware on which these models are executed. To take advantage of their full potential, it is worth assessing each subsystem's performance behavior that allows us to choose the best-fitting hardware for each domain-specific compute task. In this work, we have extended our previous work [10] with a systematic performance comparison and analysis using an empirical study to understand the performance implication due to using containerized HPC applications on x86 and OpenPOWER systems with its counterpart. These measurement results make it possible to correctly map the application kernel to the heterogeneous hardware to achieve the maximum possible performance on a particular HPC system. Moreover, it also presents recent advances on the applicability of containerization in HPC space.

Container-based HPC environment

HPC sites deploy the highly optimized environment using tunned OS, environment-specific enhanced MPI library, parallel file system, and moderately fast interconnects appropriate to the application's need. These sites are unable to fulfill the daunting set of requirements because of the dynamic nature of an almost insatiable desire for high application performance. HPC users wish their implementation environment to be highly performed, comprehensive, and coherent. It should meet the requirements of the heterogeneous workloads with complex software stacks, the elastic growth of their environmental resources, different processing elements, power-awareness, and reliability [22]. Keeping in mind several imperfections of traditional HPC sites, and research perspective on the design and development of a Container-based environment for HPC, a prototype of a Container-based HPC environment is presented in Fig. 1. Our prototype model gives a glimpse of the environmental architecture and subsystems involved and a high-level control flow model to perform benchmarking of different subsystems. Some of the essential components of the prototype are discussed below:

Table 2 Summary of related work on container-based HPC

Paper	Year	Objective	Containers Run-time	Benchmarks Used	Type of Physical Environment		Conclusion/Remarks
					Single-server	Cluster	
Migual et al. [7]	2013	performance evaluation and isolation of container-based virtualization	LXC, Linix-VServer, OpenVZ	LINPACK, STREAM, NAS Parallel-Benchmarks (NPB), IOzone, Isolation Benchmark Suite(IBS),NetPIPE	✓	✓	container-based virtualization shows near-native performance, but isolation feature requires maturity, Presented results with limited insight
David et al. [5]	2015	performance analysis of container-based solution	LXC inside virtual environment	HPL, Network Protocol Independent Performance Evaluator (NetPIPE)	✓	✗	LXC presents better performance than KVM, can be treated as a viable solution for HPC, Presented results with limited insight
Felter et al. [6]	2015	performance comparison between hypervisor-based virtualization and container-based virtualization	Docker	MySQL	✓	✗	docker equal or exceed the performance compared to KVM but need careful optimization related to I/O in both the cases, presented results with moderate analysis
Zhang et al. [17]	2015	comprehensive performance comparison of I/O virtualization between VM and container using PCI-passthrough and SR-IOV in the HPC context	Docker	IB verbs, Graph500, NAS, LAMMPS, SPEC MPI 2007	✗	✓	container-based PCI-passthrough outperform counter hypervisor-based solution, report results with moderate analysis
Arango et al. [8]	2017	performance comparison of container-based HPC environment	LXC, Docker and Singularity	HPL, IOzone, STREAM, MVA-PICH OSU Micro-Benchmarks, NAMD (NAnoscale Molecular Dynamics)	✓	✓	singularity outperform among all discussed container technology but needs improvement in the network space, reported preliminary result with very little insight
Li et al. [9]	2017	performance overhead analysis of container-based virtualization against the hypervisor-based solution	Docker	IPERF, HardInfo, STREAM, Bonnie++	✓	✗	container-based virtualization introduces less overhead compared to hypervisor-based virtualization but could be a bottleneck in storage transaction, report result with limited insight in analysis
Kuity et al. [10]	2017	Performance Evaluation of Container-Based High Performance Computing Ecosystem Using OpenPOWER	Customized Container	HPCC, and IOZone	✓	✓	container-based solution introduces less overhead compared to hypervisor-based solution in OpenPOWER System, report result with limited insight in analysis

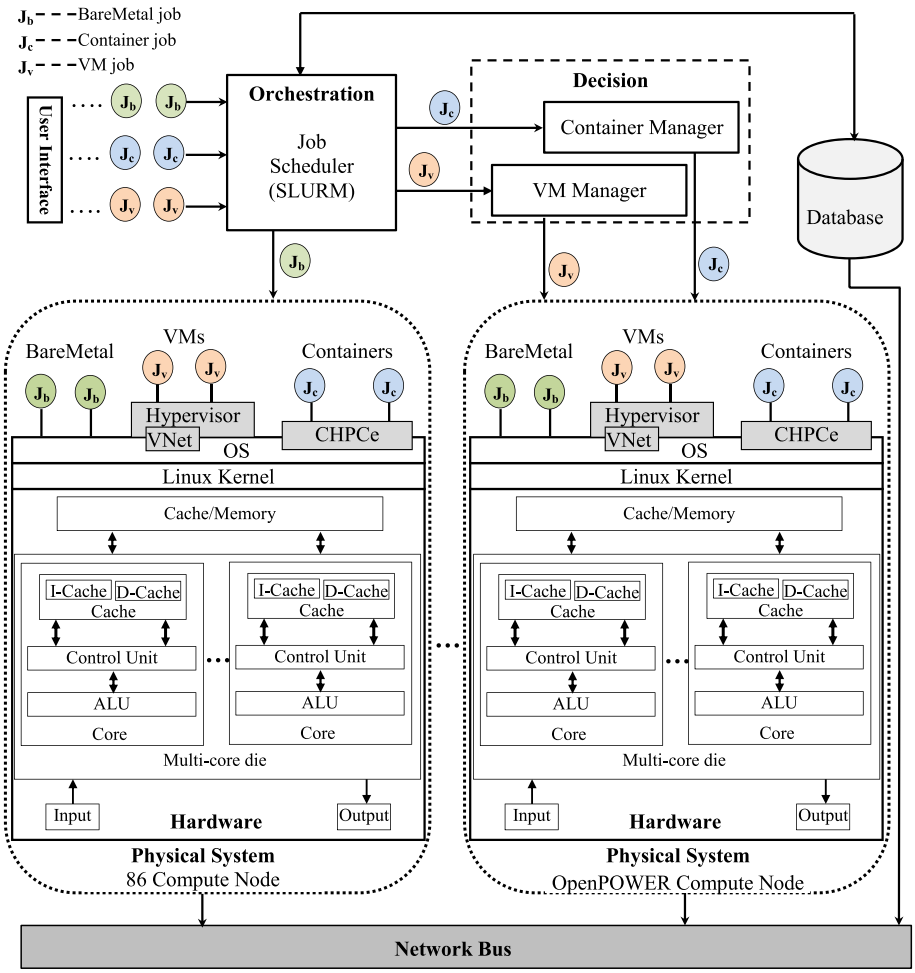

Fig. 1 High-level prototype for container-based HPC environment

User environment. In the proposed model, the image preparation takes place using the below mentioned steps:

- User prepares an image in his home environment to set up the container run-time according to the execution-time needs for the workflow application.
- User pushes the image to a private or public repository (e.g., Docker Hub).
- The prepared image can be accessed later to drag and save it to a shared location of the environment.

User interface. The user logs in to the environment using authorized credentials and submits the job to the SLURM job scheduler with the required run-time environmental parameters and the image(s) to be used.

Orchestration. The job scheduler maintains the queue of the workflow applications according to the deadline and the Quality of Service (QoS) parameter requirements. It invokes the Container manager or VM manager, depending on the tasks. It dispatches a group of related subtasks to the selected VM or a container. Database stores and processes the monitoring information. When a user submits a job specifying the resource requirements as a combination of CPU, memory, and disk constraints, the log-in node forwards the request to the workload manager. The scheduler component of the workload manager determines the nodes on which the jobs run in a distributed manner, using the first-fit decrease variant of the Vector Bin packing strategy [23]. The scheduler considers the multi-objective optimization strategy aiming to maximize resource utilization while minimizing the number of worker nodes without violating the capacity constraint of nodes. We sort the jobs in decreasing order based on the resource requirements by additive normalization to the maximum resource requirement. Then, it tries to pick the first worker node which can fulfill the requirement.

The container manager, running on the worker node, creates a set of containers to execute the job based on allocated resources on the selected node(s). When the

Table 3 System characteristics

Features Specifications	x86 System	OpenPOWER System
Processor Architecture	x86_64	ppc64le
Processor Model	Intel(R) Xeon(R) CPU E5-2630 v3	POWER8 (raw), altivec supported
No. of cores per socket	8	8
Core clock frequency (GHz)	2.400	3.857
Floating Point/clock/core	16	08
Peak Perf./core (GFlops)	38.400	30.856
L1 cache size(KB)	32+32	32+64
L2 cache size(KB)	256	512
L3 cache size(MB)	20	08
Local memory/Node (GBs)	094	128
Interconnect	10 GB/s ethernet	10 GB/s ethernet
Network Topology	Flat	Flat
Operating System	CentOS Linux release 7.3.1611 (Core)	CentOS Linux release 7.3.1611 (AltArch)
Fortran Compiler	INTEL-FORTRAN-17.0.4-196	IBM XL FORTRAN V15.1.5.1
C Compiler	INTEL COMPILER-17.0.4-196	GCC- 4.8.5
Math Library	Intel® Math Kernel Library 2017	ESSL- 5.4.0
MPI	mpich2-1.5	mpich2-1.5
Page Sizes	4KB	64KB

creation of a container cluster to perform a job is over, the container manager submits the job to the container cluster. Our investigation does not consider the dynamic scheduling of jobs with online arrival and departure over time because we focus only on a systematic performance comparison and analysis. It is outside this paper's scope. Previous work [24] reported a cHPCe model for analytical-based data locality and memory bandwidth contention-aware container placement and its performance analysis.

Decision. Container or VM manager agent facilitates the HPC application runtime environment. It chooses the most suitable compute node(s) to execute the subtask group by analyzing the collected metrics, which include available computing nodes and performance monitoring counter values. The shared image location provides a user-defined image to the container run-time.

Physical system. The container daemon, namely a container-based HPC engine (CHPCe), deploys the containers which execute a group of similar subtasks on the compute node(s).

The proposed environment finally executes the tasks on several containers following a distributed memory programming model on compute nodes and returns output to the user. The implementation details regarding the above defined environment can be found in our previous work [10].

Security aspects and limitations. The security of containerized HPC solutions mainly comes from the lack of a fine-grained access control list and users' privilege escalation. Our deployed containerized HPC environment(cHPCe) does not provide any capability beyond regular users' processes can have. When the user executes a process inside the User Defined Image (UDI), it doesn't have any elevated privileges. Images and file systems are mounted with *setuid* and *device capability* disabled conditions. These protection features combinedly provide a secure environment for containerization in the OpenPOWER HPC environment and its counterpart. However, sometimes, users may deploy their images through third-party repositories leading to several attack vectors. This can be addressed separately by introducing scanning and auditing images in the image gateway for known vulnerabilities. The application-specific security concerns are not addressed in this work explicitly. Our environment presently uses a flat network topology, which can be optimized to provide low latency and high throughput for communication-intensive applications using an intelligent routing mechanism to achieve the desired level of QoS.

The proposed work presents an architectural design of containerized HPC environment and an empirical study to assess the performance metrics of different subsystems using benchmark applications on x86 and OpenPOWER systems. To fully utilize a particular HPC system, each subsystem's performance behavior is worth assessing to choose the best-fitting hardware for each domain-specific compute task and correctly map the application kernel to the heterogeneous hardware.

Experimental testbed

This section summarizes our experimental testbed, as shown in Table 3. The smart-managed 10-Gigabit switch interconnects all the nodes. The proposed work deploys OpenStack-backed private cloud named *"Amber"* in our lab, which comprises of x86 and OpenPOWER servers. The experiments utilize OpenStack's BareMetal provisioning approach to get a BareMetal server for our experimentation. The container-based environment is formed using the namespace feature with the help of an extended resource plug-in. The plug-in interacts with the underlying BareMetal server to provision container and export Application Programming Interface(API) to the user. All the testbed employs standard cluster formation procedure to set-up the experimental environments. The VM-based testbed uses KVM hypervisor as VMM solution due to its universal acceptance and superior performance benefit. We have not imposed any optimizations during our experiments. The work tries to produce similar environments made of BareMetal(s), Container(s), and VM(s) for all the tests to achieve performance for legitimate comparison. Tuning and analysis utilities(TAU) and performance application programming interface(PAPI) tools assist us in investigating the in-depth analysis of our experimental results. TAU profiling and tracing utilizes compiler-based instrumentation techniques for our experiments.

HPC Challenge benchmark(HPCC) [25] stresses different subcomponents of the system, such as processor, memory, and interconnect to evaluate the performance of the environments made of BareMetals, containers, and virtual machines. It has seven benchmarks: HPL, STREAM, RandomAccess, PTRANS, FFTE, DGEMM, and b_eff Latency/Bandwidth. We maintain the largest problem size that fits in 70% (i.e., approximately 90GB) of the total memory for all the experiments to fulfill the standard benchmark evaluation criteria. The problem size and block size is of the order of 100000 and 100, respectively. Flat process grid ratios of 1 : 4, 1 : 8, 1 : 16, 1 : 24, 1 : 32, 1 : 48, 1 : 56, and 1 : 64 are used as input. An attempt can be made in the future to enhance the presently used flat network topology like a spine-leaf topology for organizing the switching fabric as presented [26] to improve the East-West network throughput.

Performance evaluation on HPC environments

This section critically analyses the results obtained from experiments conducted on the proposed container-based HPC environment compared to BareMetal and VM-based environments. To reduce the space usage, we discuss our profiling results using only eight or sixteen threads cases on BareMetal or container-based environment for the benchmarks. Moreover, this work avoids analyzing the hardware performance counter plot generated during experiments, unless it is necessary. This section aims to discuss the experimental results for BareMetal and container cases because VM-based cases give reduced performance due to the overhead. There are contributions in this domain, such as [5, 6, 9], which evaluate the Intel Xeon processor architecture on container space. Still, only a very few of them [10] assess the performance of the OpenPOWER system in the HPC container perspective. Therefore, the proposed work emphasizes more on the OpenPOWER system throughout our discussion. The aim of our experiments is not to achieve optimal performance from the underlying environments whereas to focus on the comparative analysis of all the three environments on both the architectures. It is essential to understand the performance of several subcomponents of a system like Compute, Memory, Interconnect, and Disk to pinpoint the bottlenecks on running the benchmark HPC applications.

To preamble more insights into our rigorous performance comparative analysis for two genres of computing systems using the containerized environment against its VM counterpart, compiler-based instrumented benchmark applications are used with TAU. TAU instrumentation lacks the support for inline assembly code, and it doesn't resolve system symbol information that comes as 'unknown' preprocessing directives. As a precautionary action, the instrumented code is throttled to reduce overhead, which has almost equal distribution for both compute systems and is ignored to highlight in further discussion to increase the accuracy of the resulting profile data. In our experiments, the median values of all the tests are considered to plot the graphs. The inner plot in every figure represents the probability density distribution of the obtained results at 32 threads over 20 repetitions of the experiments. We observe a clear peak at median values in almost all cases. The following subsections present the environment overhead and the impact of these subcomponents in detail on the HPC environment.

Time and space analysis

This experiment presents the overhead introduced to execute the basic operations of the VM and containerized HPC environments, namely startup and teardown time, along with node memory usage. The minimal "echo hello world" shell command executes inside all environments. The image gateway pulls the purposely built container image from DockerHub and VM image, converts it, and saves it into the fast shared storage location. The primary container running command "cHPCe command" is invoked to exercise the

Table 4 Overhead of executing basic operation

Overhead to execute Basic Operation	cHPCe	VM
Start Time (Sec)	0.189	103.250
Teardown Time (Sec)	0.0134	055.620

Table 5 HPL experimental parameters

Problem Size	70% of Total Available Memory
Matrices of Order	100000 × 100000
Block Size	100
Process Grids	1:04, 1:08, 1:16, 1:24, 1:32, 1:48, 1:56, 1: 64

Table 6 HPL performance

	x86			OpenPOWER		
	BareMetal	Container	VM	BareMetal	Container	VM
Achievable Peak Performance(GFlops)	783.873	782.117	619.322	213.518	212.954	203.222
% of Theoretical Peak Performance	63.71	63.65	50.40	86.50	86.27	82.33

startup and teardown overhead. The deployed containers destroy automatically when the invoking commands finish execution to ensure minimal container creation time and start executing the process as quickly as possible. We assume that Containers or VMs are ready when they can be *ssh-ed* in for our measurement. Table 4 presents the overhead to execute the essential operation in different environments. To illustrate the space usage, and memory consumption with STREAM per node is sampled at a 10-second interval before actual STREAM starts using /proc/meminfo to get insight into implementation behavior. The experimental results show that memory usage by bare metal is a median of 60.25MB with VM and cHPCe added 231.23MB and 15.4MB using proportional set size metrics, respectively.

The scheduler uses a simple 2-approximation algorithm [23] for first-fit decrease variant of the Vector Bin packing, which is NP-complete. The algorithm runs in $O(nlogn + nr)$ time using Self-Balancing Binary Search Trees, where n represents the number of jobs and r represents the number of nodes. The space complexity of the algorithm is $O(n)$, whereas the auxiliary space complexity is $O(1)$. The algorithm is well-studied and straightforward in implementation because it does not rely on system state history and has a low computational and memory footprint.

Compute performance

In this sub-section, the results of three compute benchmarks, namely High-Performance LINPACK (HPL) towards peak performance benchmark, Double-precision General Matrix Multiply(DGEMM) benchmark, and FFT benchmark are discussed.

HPL

The HPL of HPCC is used to measure the compute performance of our constructed environments on x86 and OpenPOWER systems. It solves a dense linear system of equations using LU factorization with partial row pivoting method. It reports the estimated performance of the system with the help of local matrix multiplication operations. All the LINPACK benchmark experiments use the parameters specified in Table 5 to achieve the optimal performance of our environments made of BareMetal, Container, and VM on x86 and OpenPOWER systems. Our study analyzes the achievable performance for the distributed memory computation with varying numbers of threads by dividing them equally among the compute nodes. The variation in performance is also observed by enabling the hyperthreading capability of the nodes. The environment comprises of the platform-specific libraries and the compiler to obtain optimal performance. HPL settings are fixed for all the experiments.

Figure 2 presents the performance of the HPL benchmark of three environments on x86 and OpenPOWER systems. Table 6 shows the achievable peak performance for BareMetal, container, and VM-based environments, respectively, on both x86 and OpenPOWER systems. We believe that 63.71% and 86.50% of the theoretical performance on x86 and OpenPOWER based HPC environments, respectively, reflect a reasonable performance compared to the highly engineered, purpose-built HPC Cluster.

Inference: Figure 2 shows the performance gradually degrades as the number of threads increases. The reason behind this is because the parallel jobs running on the multiple nodes introduce considerable overhead

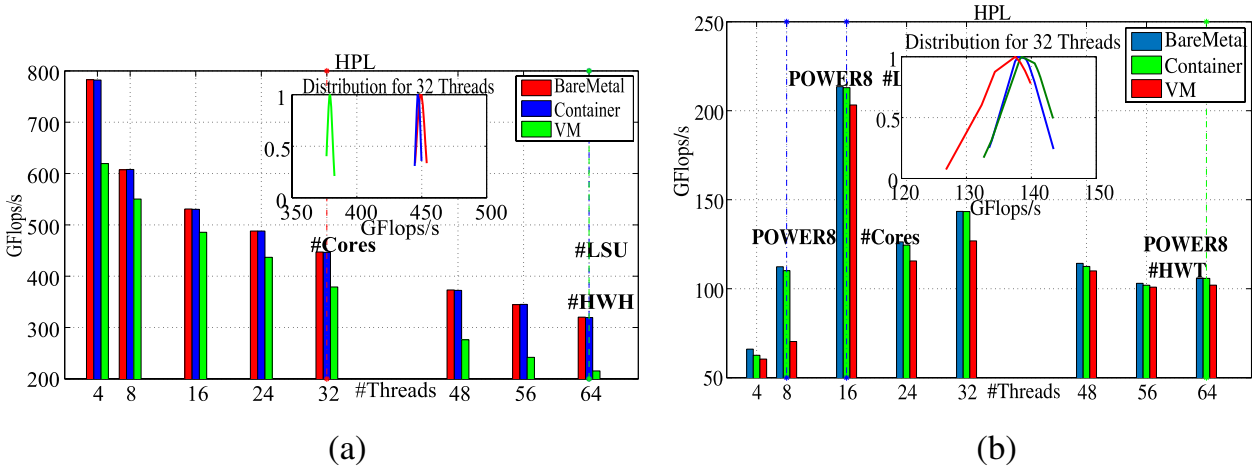

Fig. 2 Performance of HPL on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

because of the slow network connection. Figure 3 shows the network overhead due to the increase in computation threads. By profiling HPL using TAU, we observe that the time spent on synchronizing among the processes increases as the number of threads increases. In the case of HPL, the performance on the OpenPOWER system improves up to 16 threads. The reason is the OpenPOWER system has much better bandwidth compared to the used x86 system. Hence, memory contention is much less on the OpenPOWER system compared to x86. The contention is likely to occur at the last level cache; Almost two orders of magnitude performance degradation is observed as the number of computing threads doubles.

The performance of the container-based environment is nearly the same as that of the BareMetal-based environment. However, VM-based environment always shows degraded performance with an overhead up to 21% and 4.82% for x86 and OpenPOWER systems, respectively, as compared to BareMetal. In the case of a VM-based environment, KVM hides the hardware details and does not expose the CPU topology information to the workload. Therefore, the workload is unable to take advantage of this information to detect the exact nature

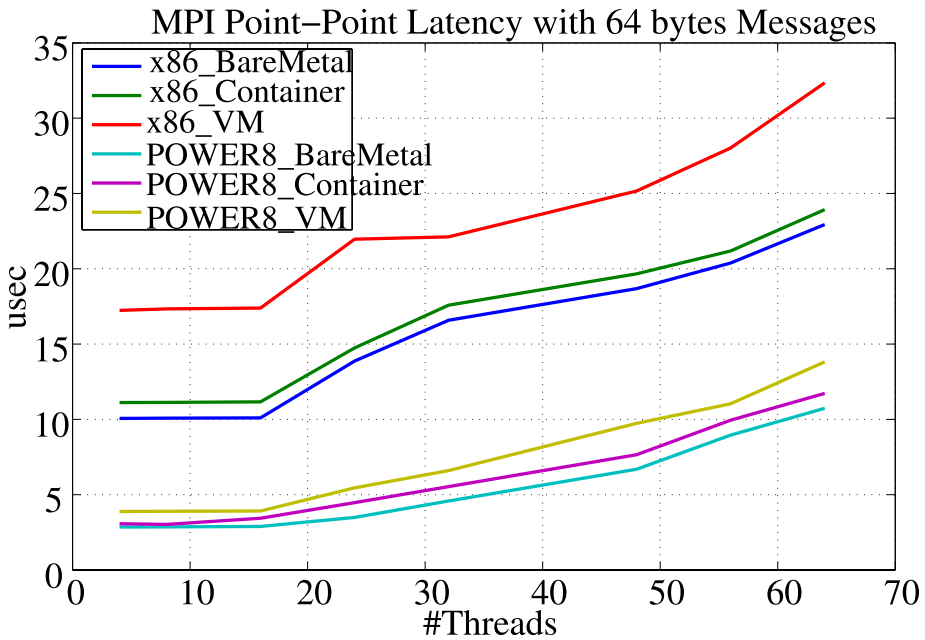

Fig. 3 Network overhead

of the system, and as a consequence, it offers much-reduced performance. Overall, all the three environments for both x86 and OpenPOWER, depict the comparable performance on the compute-intensive HPL benchmark. It is because HPL consumes most of its time in computing the well-optimized kernels that near optimally handle TLB misses, cache hierarchies, and inter-process communications. Moreover, it introduces little overhead due to the OS-level abstraction required for the container and the built-in accelerating facility incorporated at the hardware level to support virtualization efficiently.

DGEMM

A simple multi-threaded dense matrix *multiply* benchmark is used to capture the sustained floating-point computational rate of double precision real matrix-matrix multiplication of a single node. It measures the achievable double-precision FLOPS of a single node. The DGEMM benchmark uses the following computation kernels

$$C = \alpha AB + \beta C; \tag{1}$$

where A, B, and C are matrices of the dimensions $M \times K, K \times N$, and $M \times N$, respectively. In our experiments, we evaluate the performance for the matrices with the size of $10205 \times 10205, 8332 \times 8332$, and 7714×7714 to fit them into the available caches to reduce the bandwidth requirement.

Figures 4 and 5 show a comparison of the DGEMM benchmark performance among the three environments on x86 and OpenPOWER systems, respectively. Table 7 presents the achievable performances for BareMetal, container, and VM-based environments, respectively, on x86 and OpenPOWER systems.

In the case of SingleDGEMM, it is observed that the performance of the VM-based environment on both x86 and OpenPOWER systems degrades more rapidly. In the case of StarDGEMM, BareMetal and container-based environments show nearly the same performance on the x86 system. They perform better than VM-based environments up to 16 threads. However, as the number of threads starts increasing beyond this limit, all the three environments give a similar performance on the x86 system. For the OpenPOWER system, the performance increases up to eight threads, and beyond that, it shows similar characteristics, like the x86 system, for all the environments. In all the cases, the performance deteriorates as the total number of threads increases because it reduces the effective bandwidth available to each thread.

Inference: DGEMM utilizes Level-3 Basic Linear Algebra Subprograms (BLAS), having an order of N^2 data and N^3 FLOPS, in which data movement is more critical as compared to that of computation [27]. The observation regarding time spent on different functions for 16 threads on BareMetal is shown in Table 8. The experimental results report that the DGEMM benchmark spent most of its time on *MPIDI_CH3I_Progress* and *MPID_nem_tcp_connpoll* modules, and as the number of threads increases further, the time spent on both functions monotonically increases. Therefore, threads spent a decent amount of time on synchronizing and communicating among themselves. We also observe that in the case of the VM-based environment, it is magnified. DGEMM uses AVX instructions, which is slower than the typical clock speed on x86, which leads to lower performance.

Moreover, when we dig into the hardware performance counter to observe cache misses, It is found that L1 data cache misses increase as the number of threads increases. Our experiments show a worst-case comparable reduction of up to 18.57% in L3 data cache misses in OpenPOWER w.r.t counterpart due to the OpenPOWER processor has a dedicated 8MB L3 cache per core. In contrast, the x86 processor has a total of 25MB. The OpenPOWER system shows less cache misses compared to the x86 system. The major hindrance is still the data movement to effectively mitigate the difference in memory

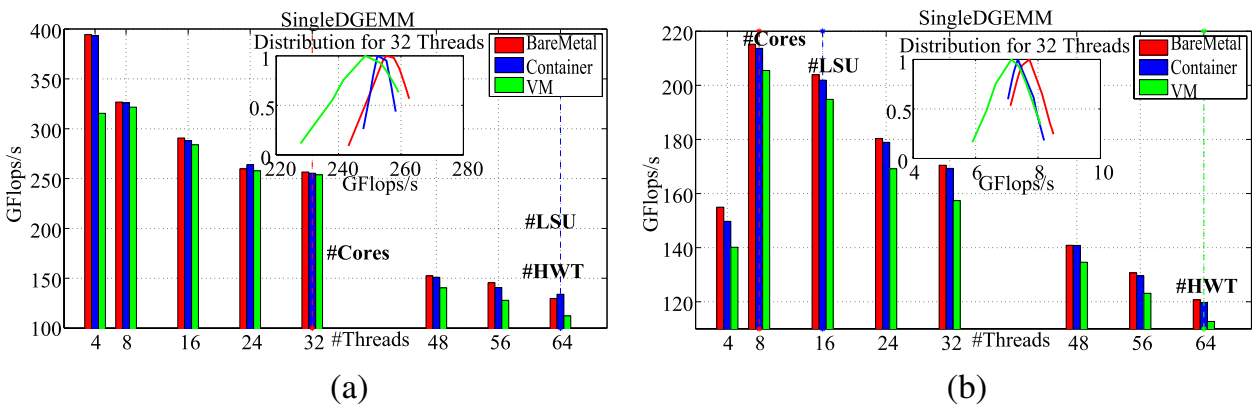

Fig. 4 Performance of SingleDGEMM on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

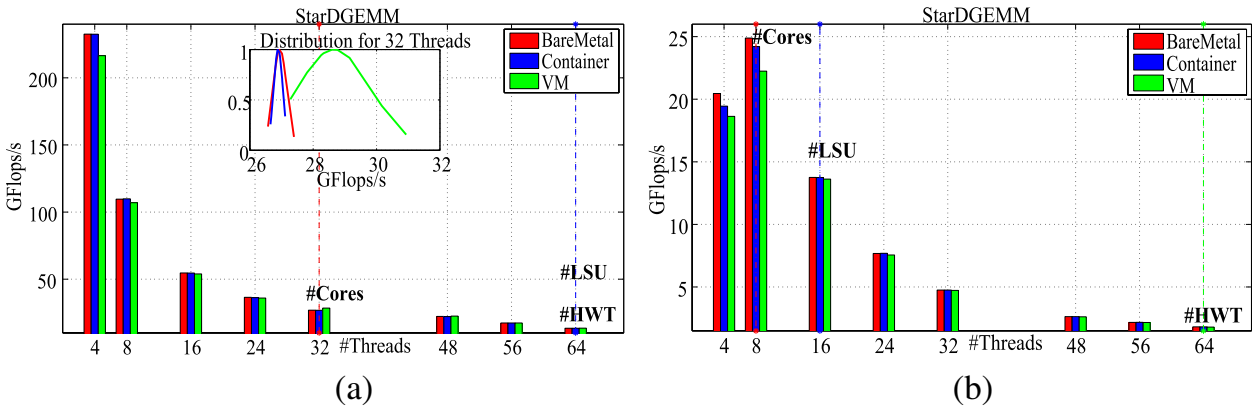

Fig. 5 Performance of StarDGEMM on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

Table 7 SingleDGEMM Performance

	x86			OpenPOWER		
	BareMetal	**Container**	**VM**	**BareMetal**	**Container**	**VM**
Achievable Peak Performance(GFlops)	394.444	393.523	315.494	215.177	213.646	205.476
% of Theoretical Peak Performance	64.20	64.05	51.35	87.17	86.55	83.24

Table 8 Accumulated exclusive time per process of SingleDGEMM in case of 16 threads

Processes	**Interpretation**	**x86**		**OpenPOWER**	
		Exclusive Execution Time (Sec)	**% of Total Execution Time**	**Exclusive Execution Time (Sec)**	**% of Total Execution Time**
MPIDI_CH3I_Progress	maintain process progress state for asynchronous communications	1,436.507	65.74	1,991.025	45.11
MPID_nem_tcp_connpoll	tcp communication between processes	0424.685	19.44	1,812.856	41.07
mkl_blas_avx2_dgemm_kernel_0	dgemm function of the Math Kernel Library from Intel	0127.769	05.85	-	-
MPID_nem_network_poll	network polling for communicating threads	-	-	0339.154	00.08
Unknown	not reported	0071.447	03.27	0149.722	00.03
MPIDU_Sched_are_pending	Check if all_schedules is empty	0062.728	02.87	0121.134	00.03
Others	for rest	0004.303	00.20	-	-

latencies and bandwidth in the memory hierarchy to achieve high performance. This behavior is noticeable only when the number of threads increases more than the number of the physical cores on both the systems. The optimal usage of the cache system reduces the total number of memory accesses, which increases the availability of the memory system for other processors.

In DGEMM, the blocking also plays a vital role in the achievable performance by minimizing the eviction of data from the cache, which helps to effectively use the cache hierarchies to hide the performance disparity between memory and the processor on both multi-core and many-core architectures. We observe worst-case performance variation of 49.23% and 37.57%, respectively, for x86 and OpenPOWER system between the performance of the worst cache block size and the best cache size with diagonal matrices of size 4k, 8k, 10k, 16k, and 32k. The commission reported is with a 90%

confidence interval. As the OpenPOWER system has a rich cache subsystem compared to the x86 system, the OpenPOWER system, taking advantage of the higher memory bandwidth, keeps a significant portion of the data in the fast end of the memory for reuse during computation. It helps to achieve significantly closer performance compared to the theoretical peak.

Finally, the tile dimensions used in the experiment need to be selected carefully, keeping the architecture in mind because it may reduce the performance due to TLB pressure and cache associativity conflict. To maximize the performance of DGEMM on OpenPOWER, we can also impose the instruction level optimization to take advantage of the dual instruction pipeline.

FFT benchmark

This benchmark is used to measure the floating-point rate of execution of double-precision complex one-dimensional discrete Fourier transform (DFT) of size *m*. It stresses on the inter-process communication using large messages. Figures 6 and 7 show the performance of the FFT benchmark for 4 to 64 threads for each of the three environments on x86 and OpenPOWER systems. The performance of BareMetal and container-based environments are quite similar. The VM-based environment always depicts reduced performance compared to the other two instances in all the experiments. The performance degrades in all cases as the number of threads increases due to the high latency and lower bandwidth availability of intra-node communication.

Inference: FFT algorithm can reduce or eliminate the time spent in the communication mode. It is possible by overlapping computation on a current plane with the communication on the previously processed plane depending on the computation and communication characteristics of the underlying machine [28]. During 1D FFT computation, each data point must be loaded and stored from memory at least once, in which the cache is exploited to prevent additional memory access within each node for each operand of the floating-point operation. Similar cache behavior is observed for both OpenPOWER and x86 cases. Table 9 presents the time spent on different functions for eight threads on BareMetal based environment. In the case of MPIFFT,

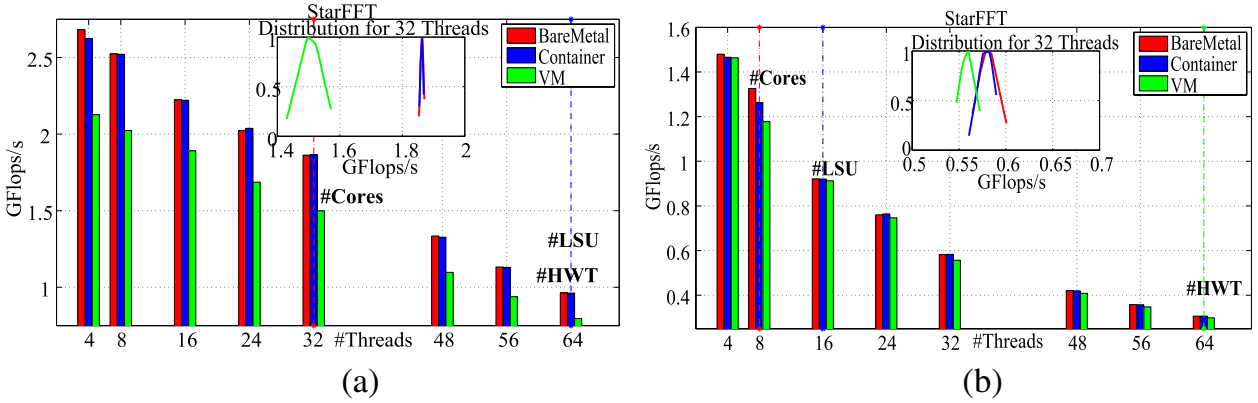

Fig. 6 Performance of StarFFT on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

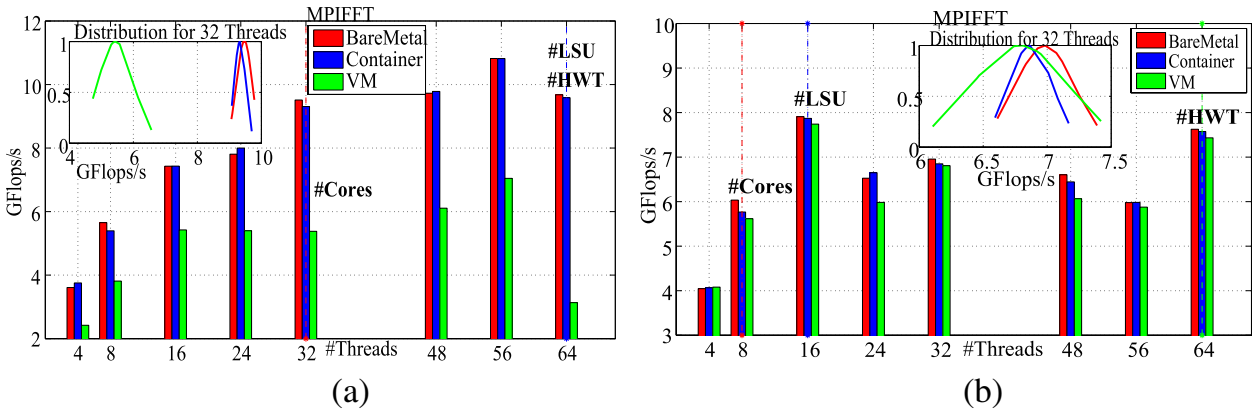

Fig. 7 Performance of MPIFFT on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

Table 9 Accumulated exclusive time of processes of MPIFFT in case of 8 threads

Processes	Interpretation	x86		OpenPOWER	
		Exclusive Execution Time(Sec)	% of Total Execution Time	Exclusive Execution Time(Sec)	% of Total Execution Time
HPCC_bcnrand	IEEE 64-bit floating-point pseudorandom generator	-	-	2.828	16.27
Tau_global_getLightsOut:TauCAPI.cpp	to control disable profiling	-	-	1.704	09.81
FunctionInfo::IsThrottled() const:FunctionInfo.h	throttled timers	0.251	03.43	1.634	09.40
MPIDI_CH3I_Progress	maintain process progress state for asynchronous communications	1.032	14.10	1.254	07.22
Tau_get_thread:TauCAPI.cpp	Return id of the worker thread	0.172	02.35	1.174	06.76
Tau_memory_wrapper_ enable:TauMemory.cpp	memory wrapper enable library	-	-	0.915	05.27
MPID_nem_tcp_connpoll	tcp communication between processes	0.924	12.63	0.797	04.59
MPID_nem_network_poll	network polling for communicating threads	0.203	02.77	-	-
MPID_nem_barrier	thread barrier for synchronization	0.157	02.15	-	-
Unknown	not reported	1.433	19.59	0.657	03.78
Tau_memory_wrapper_ disable:TauMemory.cpp	memory wrapper disable library	0.137	01.87	0.593	03.41
Tau_lite_start_timer:TauCAPI.cpp	start timer for a phase	-	-	0.532	03.06
HPCC_pzfft1d	parallelized FFT codes	-	-	0.443	02.55
HPCC_PoolReturnObj	return pool of processes represented by buckets	0.265	03.62	-	-
HPCC_ra_Heap_IncrementKey	Random access heap key increment	0.223	03.05	-	-
HPCC_InsertUpdate	Each process (PE) maintains a set of destination PE	0.180	02.46	-	-
HPCC_GetUpdates	maintains the updates for each PE	0.102	01.39		
Tau_lite_stop_timer:TauCAPI.cpp	stop timer for a phase	0.419	05.73	0.431	02.48
Tau_profile_c_timer:TauCAPI.cpp	profile timer in C			0.394	02.27
Tau_global_incr_insideTAU:TauCAPI.cpp	global increment inside TAU	0.317	04.33	0.380	02.19
Tau_global_decr_insideTAU:TauCAPI.cpp	global decrement inside TAU	0.486	06.64		
dddiv:bcnrand.inst.c	IEEE 64-bit floating-point pseudorandom generator	-	-	0.378	02.18
RtsLayer::TheEnableInstrumentation():RtsLayer.cpp	To enable instrumentation	-	-	0.370	02.13
Tau_lite_start_timer:TauCAPI.cpp	start timer for a phase	-	-	0.325	1.87
Tau_global_getLightsOut:TauCAPI.cpp	to control disable profiling	0.194	02.65	0.299	1.72
FunctionInfo::IsThrottled() const:FunctionInfo.h	throttled timers	-	-	0.277	1.59
fft8:fft235.inst.c	Radix-8 fft routine	-	-	0.231	1.33
Tau_lite_stop_timer:TauCAPI.cpp	stop timer for a phase	0.184	02.51	0.226	1.3
Tau_start_timer:TauCAPI.cpp	start timer for a particular thread	0.200	02.73	0.207	1.19
Tau_global_incr_insideTAU:TauCAPI.cpp	global increment inside TAU	-	-	0.164	0.94
Tau_get_thread:TauCppAPI.h	Return id of the worker thread	-	02.35	0.128	0.74
Others	for rest	0.183	02.50	-	-

we see that FFT experiment spent a good amount of the time(10% to 30% of the total time) on *HPCC_bcnrand* procedure which generates a sequence of IEEE 64-bit floating point pseudo-random number. Further, the time spent on the *MPIDI_CH3I_Progress* increases gradually as the number of threads increases, which indicates that the time spent on asynchronous communications could be a limiting factor in the performance

of the FFT. The OpenPOWER system has fully associative 64-entry Instruction Effective to Real Address Translation Table (IERAT). Therefore, at any time, it can translate 64 virtual pages. Hence, it is important to choose the optimal block size of the problem to minimize TLB misses and fully exploit the cache hierarchy.

It can be concluded that the critical design consideration for the FFT benchmark depends not on the absolute per-processor performance, but the relative balances on the per-node compute capacity, intra-node bandwidth, and inter-node bandwidth.

Memory performance

In this sub-section, the results of two memory-stress benchmarks, namely STREAM and RandomAccess, are discussed.

STREAM

A simple synthetic benchmark application used to evaluate the sustainable memory bandwidth of the constructed environments. It measures the computation rate using four simple vector kernels.

$$\begin{aligned} COPY &: c = a, \\ SCALE &: b = \alpha c, \\ ADD &: c = a + b, \\ TRIAD &: a = b + \alpha c. \end{aligned} \tag{2}$$

where a, b, c are vectors and α is a scalar.

The STREAM benchmark experiment is done with an array size, which requires a total memory of more than 2x the size of L3 caches for all the environments on both x86 and OpenPOWER systems.

Figures 8 and 9 display the sustainable memory bandwidth using the STREAM benchmark for each of the three environments on x86 and OpenPOWER systems. The average achievable memory bandwidth running STREAM Triad kernel is up to 13.14125 GB/s, 13.14915 GB/s, and 11.6643 GB/s for BareMetal, container, and VM-based environments on x86 systems, respectively. However, OpenPOWER based environments result up to 23.9111 GB/s, 24.9417 GB/s, and 24.84495 GB/s for

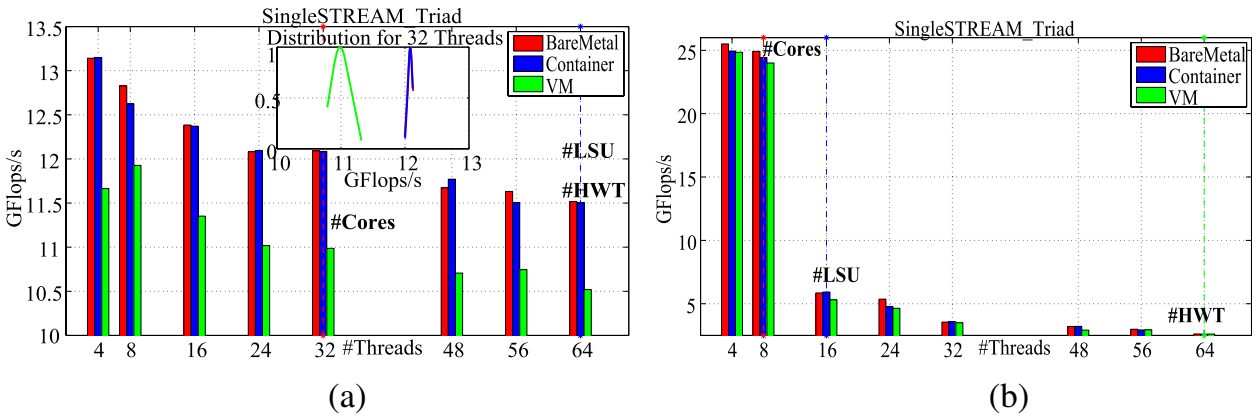

Fig. 8 Performance of SingleSTREAM_Triad on BareMetal-, Container-, and VM-based environments on (a) x86. (b) OpenPOWER

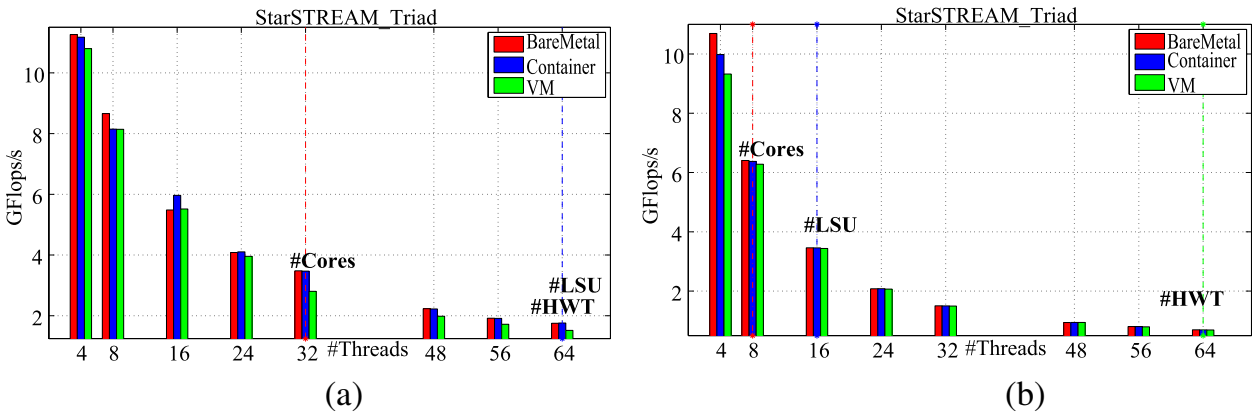

Fig. 9 Performance of StarSTREAM_Triad on BareMetal-, Container-, and VM-based environments on (a) x86. (b) OpenPOWER

BareMetal, container, and VM cases, respectively. In the case of SingleSTREAM, VM-based environment shows reduced bandwidth compared to BareMetal-based environments. The container-based environment behaves nearly similar to the BareMetal case on x86 systems. A negligible difference is observed between the performance of all three environments on the OpenPOWER system.

Inference: The OpenPOWER processor has two symmetric load pipelines and two load/store pipelines. Moreover, it has interfaces to the external memory with an integrated cache that offers twice the number of load requests than the stores [13]. The achievable bandwidth for copy, scale, and the sum is lower than the triad. The triad uses two loads and one store stream to fit the memory links. In the OpenPOWER-based environment, the GCC compiler fails to use the vector instruction for the copy benchmark. The Stream benchmark offers the peak achievable bandwidth with four threads, at which point every LSU is busy. In the SingleStream case, only one thread performs the computation, and the entire bandwidth is available for it. The performance degradation in the VM-based environment is due to the address translation overhead, and double caching problem in the virtualization layer.

In the case of StartSTREAM, nearly equal performance is observed for BareMetal, container, and VM-based environments on x86 and OpenPOWER systems. All threads perform computation, so all the participating threads share the available bandwidth and L3 cache among themselves. All the experiments show degraded performance as the number of threads starts increasing due to memory contention. The efficiency of the prefetch mechanism has a high impact on the STREAM benchmarks performance due to its highly regular nature. Moreover, only the first and third level cache affects the STREAM benchmark. As a part of the prefetch stream, it fetches up to six cache lines into the L1 and L2 cache, and 16 lines into L3 cache ahead of the stream. Although the prefetch request may miss the L2 cache as it has the same prefetch line as L1, it will surely hit the L3 cache as it is ahead of the L1 cache.

We analyze hardware performance counters to understand the impact of the cache hierarchy on the result. It is clear from the hardware performance counter data that instruction and data cache misses gradually decrease as the number of threads increases. Moreover, the observations show that the time spent in synchronization amongst the threads increases with an increased number of threads. Table 10 depicts the time spent by different functions. Figure 10 shows the cache behavior from hardware performance values during StarSTREAM benchmark experiments on the container-based environment.

From the above discussion, It can be concluded that the higher density of threads and poor choice of the locality can negatively impact the memory performance, which indicates that the thread binding has a crucial role to achieve performance on the OpenPOWER system which has not been imposed in our experimentation to maintain consistency in performance result.

RandomAccess benchmark

This benchmark assesses the peak capacity of the memory subsystem by updating the random location of the system memory. The benchmark works on a large

Table 10 Accumulated exclusive time of processes of StarStream in case of 16 threads

Processes	Interpretation	x86		OpenPOWER	
		Exclusive Execution Time(Sec)	% of Total Execution Time	Exclusive Execution Time(Sec)	% of Total Execution Time
tuned_STREAM_Add._omp_fn.5:stream.inst.c	OpenMP implementation of Stream add kernel	224.717	25.29	-	-
tuned_STREAM_Triad._omp_fn.6:stream.inst.c	OpenMP implementation of Stream triad kernel	224.553	25.27	-	-
tuned_STREAM_Copy._omp_fn.3:stream.inst.c	OpenMP implementation of Stream copy kernel	169.924	19.12	-	-
tuned_STREAM_Scale._omp_fn.4:stream.inst.c	OpenMP implementation of Stream scale kernel	169.681	19.09	-	-
HPCC_Stream._omp_fn.1:stream.inst.c	HPCC parent stream function	056.604	06.37	601.192	59.78
Unknown	not reported	-	-	170.243	16.93
MPIDI_CH3I_Progress	maintain process progress state for asynchronous communications	-	-	100.768	10.02
MPID_nem_tcp_connpoll	tcp communication between processes	-	-	085.760	08.53
computeSTREAMerrors	compute the average errors for each array	020.056	02.26	020.940	02.08

distributed table of size 2^p, occupying approximately half of the system memory and profiles the memory architecture of the system. MPIRandomAccess uses the main table of size 8589934592(2^{33}) words in all the tests. The PE main table of size 2^{31}, 2^{30} , 2^{29}, 2^{33}/24, (2^{33})/48, (2^{33})/56, and (2^{28}) words are used respectively for 4, 8, 16, 24, 32, 48, 56, 64 threads in the case of both MPIRandomAccess and StarRandomAccess. It performs four times the total number of updates.

Figures 11 and 12 capture the performance of the RandomAccess benchmark for BareMetal, container, and VM-based environments on x86 and OpenPOWER systems. We observe that the performance of the StarRandomAccess on x86 systems increases as the number of threads increases up to 16 in all the three cases. Further, it gradually decreases as the number of threads increases. For the OpenPOWER system, container-based and BareMetal-based environments show similar performance. The VM-based environment introduces overhead up to 61.5% and 1.74% as compared to the BareMetal-based environment on x86 and OpenPOWER systems, respectively. In the case of MPIRandomAccess, we notice that the performance increases as the number of threads increases up to 32, and beyond that, it starts decreasing on the x86 systems. For the OpenPOWER system, the experimental results show that the performance of all the three environments increases up to 24 threads, and further, it drops sharply. The VM-based environment presents up to 32.4% and 12.64% overhead as compared to the BareMetal-based environment on x86 and OpenPOWER systems, respectively, due to high pressure on TLB and

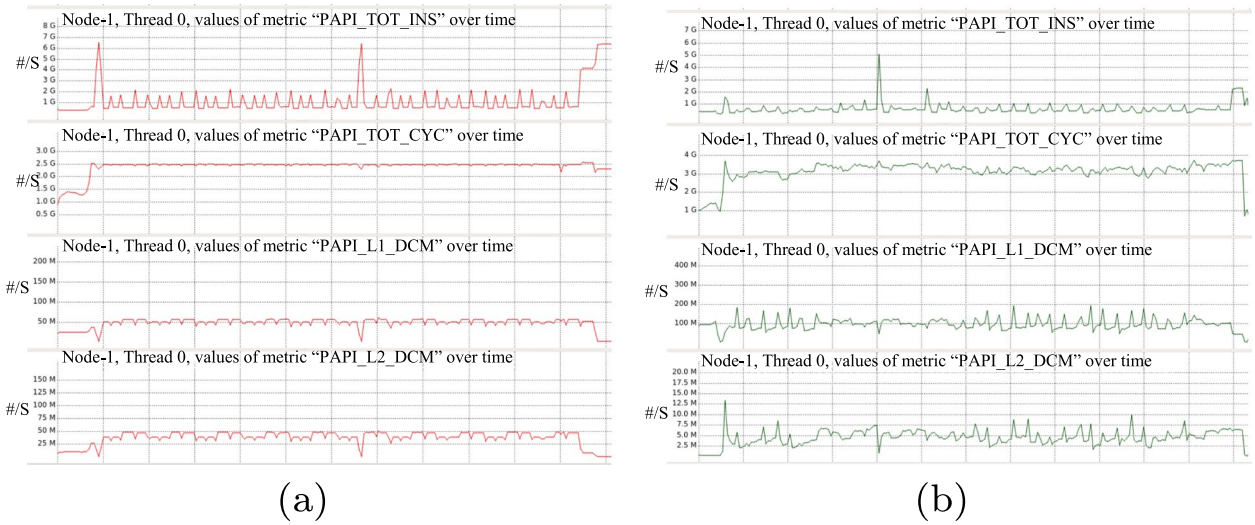

Fig. 10 Performance Counter values for 16 threads on (**a**) x86. (**b**) OpenPOWER

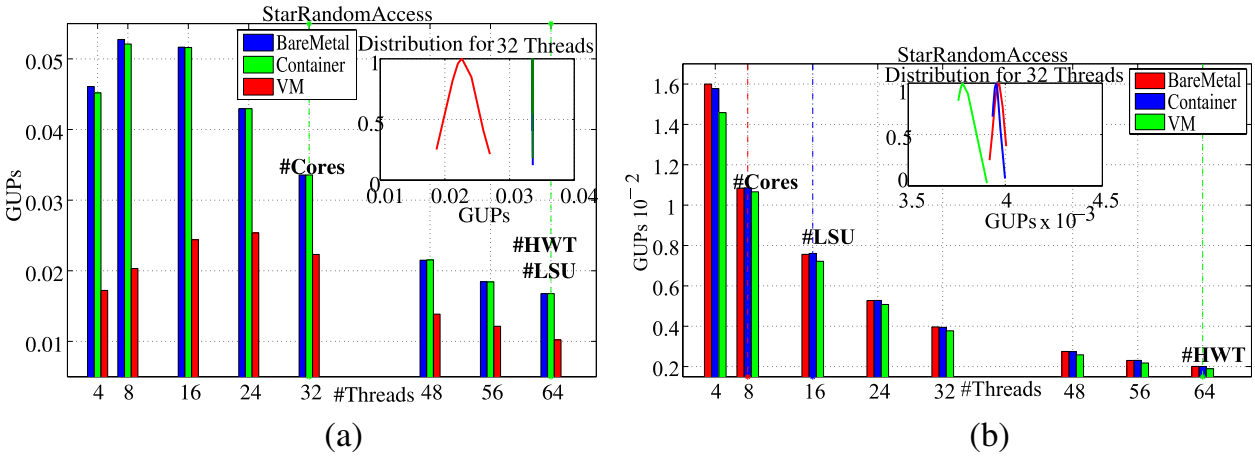

Fig. 11 Performance of StarRandomAccess on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

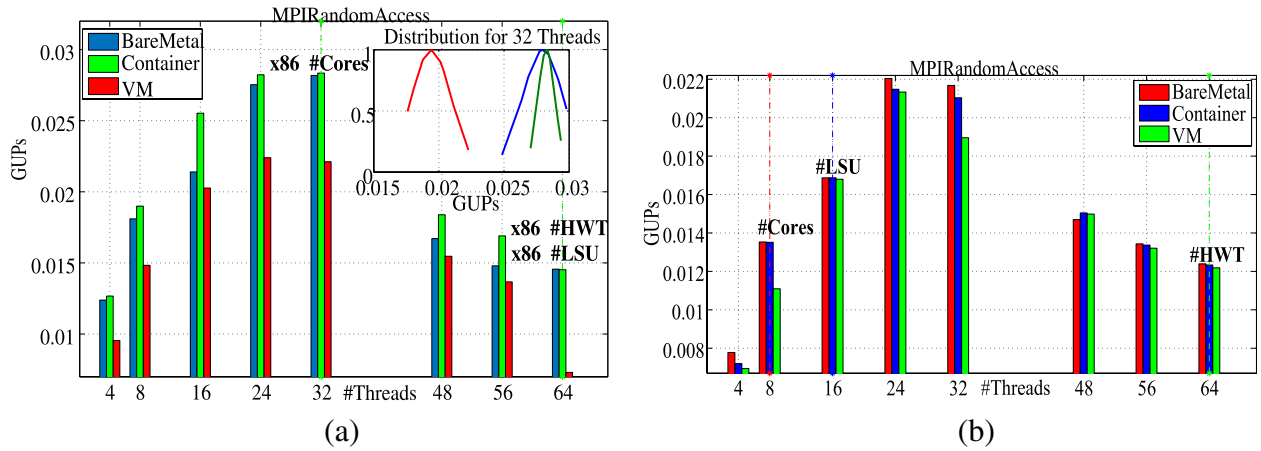

Fig. 12 Performance of MPIRandomAccess on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

handling of the substantial number of short messages between the threads in a virtualized environment.

Inference: The RandomAccess Benchmark is designed to exhibit very low temporal and spatial locality, which updates random entries in a large table that is unlikely to be cached [29]. In a distributed memory model, the total available memory is partitioned among the nodes. Moreover, within a node, only a fraction of the entire node memory is local to a particular thread. The MPIRandomAccess benchmark carries out local as well as remote update of a large globally distributed table to the whole system memory in parallel. It allows us to look ahead and store at most 1024 updates before going to update in the table entries. The mapping mode 1 of Random Access benchmark shows the best case performance, which perfectly balances the workload on each core. The table on each core is allocated on a bank that has NUMA affinity, thereby reducing cache consistency overhead and inter-node traffic. Hence, the performance of the benchmark depends on the choice of the mapping and optimization strategy to the cores of a multi-core node.

The performance decreases as the number of threads increases in all the cases after 32 threads. It is due to bi-section bandwidth and an average number of bytes sent over the network to update the table locations. Our experiments use the simple flat network for communication among the nodes. As the number of threads increases due to an increase in some bytes over the network, the bi-section bandwidth eventually becomes the bottleneck for the performance. We observe, by profiling the benchmark as shown in Table 11 for eight threads, that the benchmark spends a reasonable amount of time in sending and receiving the messages. Suppose the global table maintained by the RandomAccess benchmark is not a power of two. In that case, computing the node id and offset of the table update location requires an integer division operation. It is a maximum of twice the order of magnitude than the other operations. It significantly influences the performance of the benchmark.

We conclude that the performance of the benchmark depends on the bi-section communication bandwidth, multi-core mapping strategy, and performance of the integer division operation for non-power of two-node cases.

Interconnect performance

In this sub-section, the results of two interconnect-related benchmarks, namely PTRANS (Parallel Matrix transpose) and b_eff are discussed.

PTRANS

This benchmark is used to inspect the total communication capacity of the system interconnect. It exchanges a large number of messages simultaneously between each pair of processors to investigate the communication capacity. A matrix of size 50000 x 50000 processes, and process grids of size 1 x 4, 1 x 8, 1 x 16, 1 x 24, 1 x 32, 1 x 48, 1 x 56, and 1 x 64 are taken as an input.

Figure 13 captures the performance of the PTRANS benchmark for BareMetal, container, and VM-based environments on x86 and OpenPOWER systems. In this benchmark, we discuss the result for the optimal configuration only. The performance of the benchmark scales very well as the number of threads increases. However, it shows different behavior for 48 threads for BareMetal and container-based environments on x86 systems. The performance of the VM-based environment increases as the number of threads increases up to 32, after that, it starts to decline. However, it offers much-degraded performance with a maximum

Table 11 Accumulated Exclusive time of processes of MPIRandomAccess in case of 8 threads

Processes	Interpretation	x86		OpenPOWER	
		Exclusive Execution Time(Sec)	% of Total Execution Time	Exclusive Execution Time(Sec)	% of Total Execution Time
Unknown	not reported	1.051	37.73	1.433	19.59
MPIDI_CH3I_Progress	maintain process progress state for asynchronous communications	0.554	19.90	1.032	14.10
MPID_nem_tcp_connpoll	tcp communication between processes	0.112	04.02	0.924	12.63
Tau_global_decr_insideTAU:TauCAPI.cpp	global decrement inside TAU	-	-	0.486	06.64
Tau_global_incr_insideTAU:TauCAPI.cpp	global increment inside TAU	0.026	00.93	0.317	04.33
HPCC_PoolGetObj	return pool of processes represented by buckets	0.311	11.17	-	-
HPCC_PoolReturnObj	return pool of processes represented by buckets	0.173	06.21	0.265	03.62
FunctionInfo::IsThrottled() const:FunctionInfo.h	throttled timers	-	-	0.251	03.43
HPCC_ra_Heap_IncrementKey	Random access heap key increment	-	-	0.223	03.05
MPID_nem_network_poll	network polling for communicating threads	-	-	0.203	02.77
Tau_start_timer:TauCAPI.cpp	start timer for a particular thread	0.200	02.73	0.200	02.73
Tau_global_getLightsOut:TauCAPI.cpp	to control disable profiling	0.194	02.66	0.194	02.65
Tau_lite_stop_timer:TauCAPI.cpp	stop timer for a phase	0.129	04.63	0.419	05.73
Tau_lite_stop_timer:TauCAPI.cpp	stop timer for a phase	0.079	02.84	0.184	02.51
HPCC_InsertUpdate	maintains the updates for each PE	-	-	0.180	02.46
Tau_get_thread:TauCAPI.cpp	Return id of the worker thread	-	-	0.172	02.35
MPID_nem_barrier	thread barrier for synchronization	-	-	0.157	02.15
Tau_memory_wrapper_disable:TauMemory.cpp	memory wrapper disable library	0.135	04.85	0.137	01.87
HPCC_GetUpdates	Each process (PE) maintains a set of destination PE	-	-	0.102	01.39
PMPI_Testany	used to wait for the completion of one out of several operations	0.074	02.77	-	-

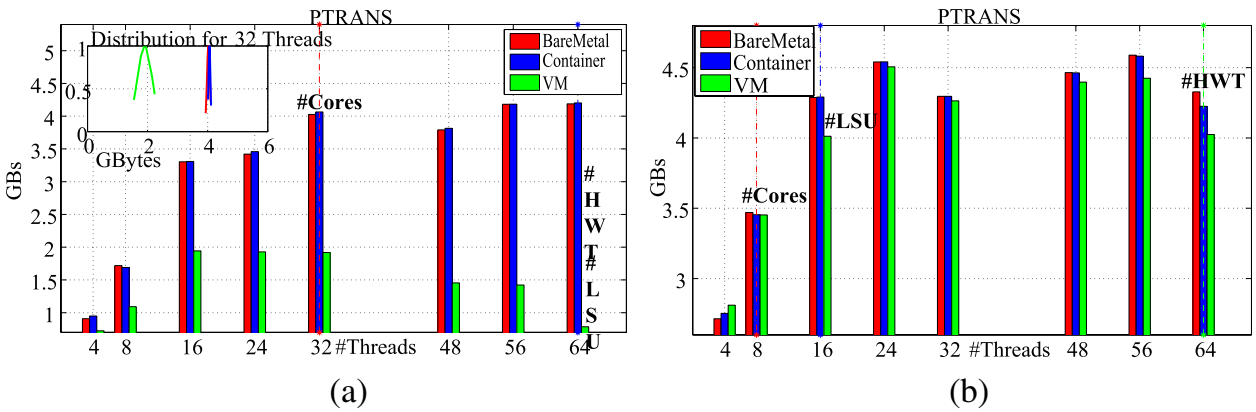

Fig. 13 Performance of PTRANS on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

overhead up to 81.26% and 81.32% compared to that of BareMetal and container-based environment, respectively, on the x86 systems.

In the case of the OpenPOWER system, the performance of all three environments boosts as the number of threads increases up to 24, and further, it begins to

decrease. It shows different behavior for 56 threads. The BareMetal and container-based environment, showing a similar performance, outperform the VM-based environment in almost all the cases on the OpenPOWER system. The benchmark displays the diminishing performance as the number of threads increases because it stresses the global network and a high number of threads involvement in communication through interconnect. However, Fig. 13 shows entirely different behavior in the case of x86 based environments.

b_eff benchmark

This benchmark captures the effective bandwidth and latency of the interconnect of the environment. It exchanges 8 bytes and 2,000,000 bytes of messages to measure latency and bandwidth of the communication interconnect, respectively, using simple MPI point-point routines. In the case of ring communication, all the threads arrange themselves in a ring fashion, and each thread sends and receives messages tofrom its neighbors in parallel. It utilizes the number of threads like 4, 8, 16, 24, 32, 48, 56, and 64. The Ping Pong communication uses 992(i.e.,32*(32-1)), 2256, and 3080 pairs of threads for latency and bandwidth measurement.

Latency: Figures 14 and 15 plot avgPingPong and random-order ring latency for BareMetal, container, and VM-based environments on x86 and OpenPOWER systems. The BareMetal and Container-based environments report a similar behavior for avgPingPong and random-order latency on the x86 system. For the VM-based environment, latency increases as the number of threads increases in random-order ring latency on the x86 systems. It is due to the virtualization overhead, for the increasing number of threads. However, in the avgPingPong case, the latency decreases as the number of threads increases up to 24; after that, it starts increasing. The reason behind this is not evident to us. In all the cases, usually, latency boosts as the number of threads increases due to the required number of multi-stage traversal by the threads. All the three environments show excellent scaling as the number of threads increases for both avgPingPong and random-order ring latency on

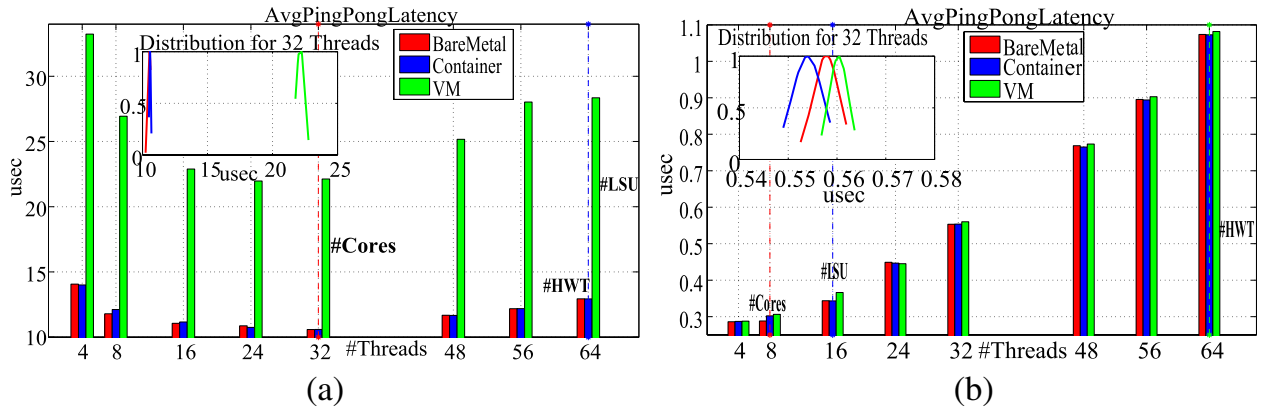

Fig. 14 Performance of AvgPingPongLatency on BareMetal-, Container-, and VM based environments on (**a**) x86. (**b**) OpenPOWER

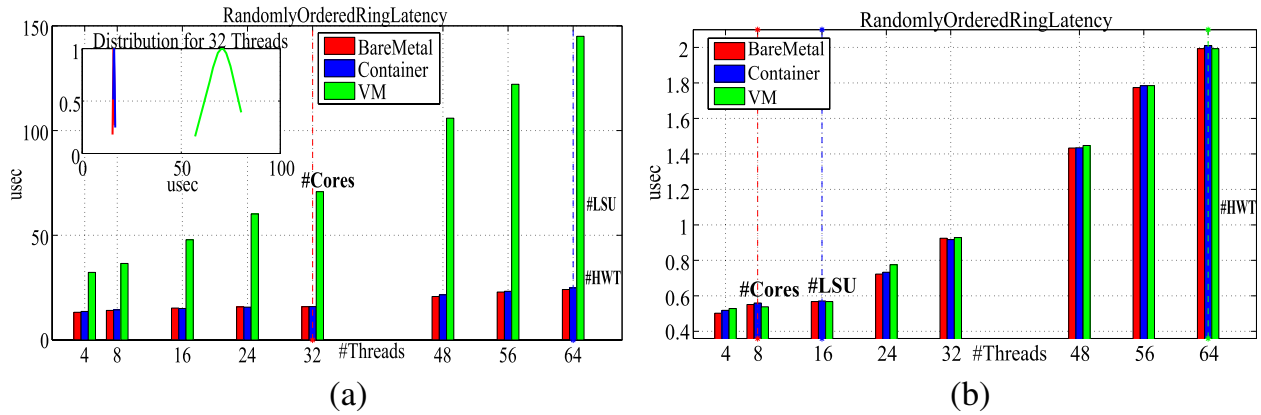

Fig. 15 Performance of RandomlyOrderedRingLatency on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

the OpenPOWER system. Both the benchmarks depict nearly a similar performance in all three cases with slight variation in latency with VM-based environment on the OpenPOWER system.

Inference: On ping-pong communication, only one pair of processes communicate with each other using MPI standard blocking send and receive call. At the same time, all the other processes wait on a blocking receive call to avoid any potential interference. The profiling of e_bff benchmark is done to observe the time spent on different functions, as shown in Table 12. Communication latency would cause an increase in latency in all cases. Still, the impact of extra communication costs reduces the total communication performance as the number of threads increases. Figure 15 depicts that as the number of threads increases, the average latency increases significantly in the case of x86 based environment as compared to the OpenPOWER based environment. It is because, in the case of OpenPOWER based environment, all the communications happen within the same node with extremely low latency. However, in the case of x86 based environment, although half of the communication occurs within the node, but as the number of threads increases, most of the threads communicate through a router with relatively higher latency, which leads to the increase in overall latency.

Bandwidth: Figures 16 and 17 display the avgPingPong and random-order ring bandwidth for BareMetal, container, and VM-based environments on x86 and OpenPOWER systems. Nearly similar performance behavior is observed for both avgPingPong and random-order ring bandwidth on BareMetal and container-based environments using x86 and OpenPOWER systems. The random-order ring bandwidth shows excellent scalability with a proportional decrease in the bandwidth as the number of threads increases for all the three environments on the x86 systems. The VM-based environment on x86 systems depicts noticeable performance overhead, as expected, due to the virtualization overhead. In the OpenPOWER case, all three environments display diminishing performance as the number of threads increases, with VM-based environment showing the degraded performance compared to the other two, up to 24 threads, and beyond that, all converge. In the case of

Table 12 Accumulated exclusive time of processes of b_eff in case of 16 threads

Processes	Interpretation	x86		OpenPOWER	
		Exclusive Execution Time(Sec)	% of Total Execution Time	Exclusive Execution Time(Sec)	% of Total Execution Time
MPIDI_CH3I_Progress	maintain process progress state for asynchronous communications	0051.736	50.07	2139.827	46.30
Unknown	not reported	0025.976	25.14	0108.248	02.34
MPID_nem_tcp_connpoll	tcp communication between processes	0019.023	18.41	1808.054	39.12
MPID_nem_network_poll	network polling for communicating threads	0003.046	02.95	-	-
lmt_shm_recv_progress:mpid_nem_lmt_shm.c	message receiving mechanism of MPI	-	-	0068.265	01.48

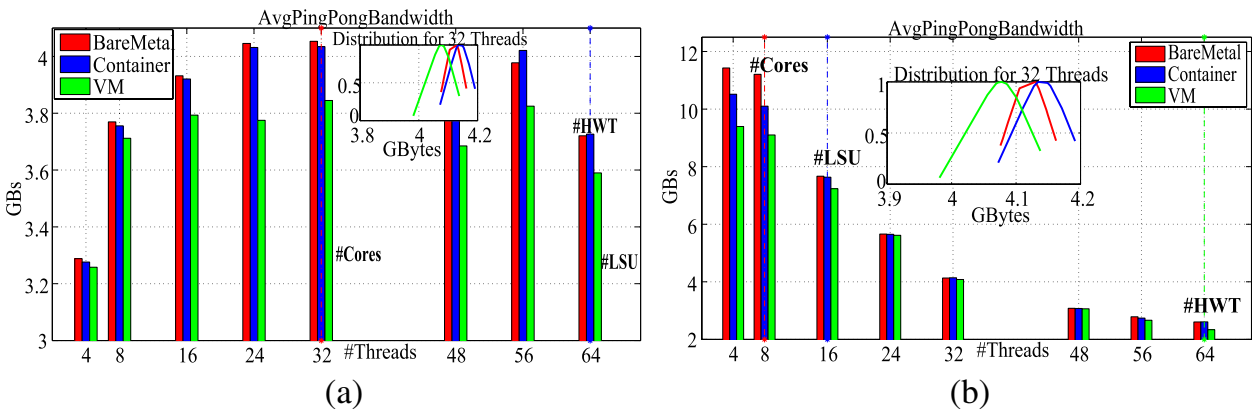

Fig. 16 Performance of AvgPingPongBandwidth on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER. RandomlyOrderedRingBandwidth

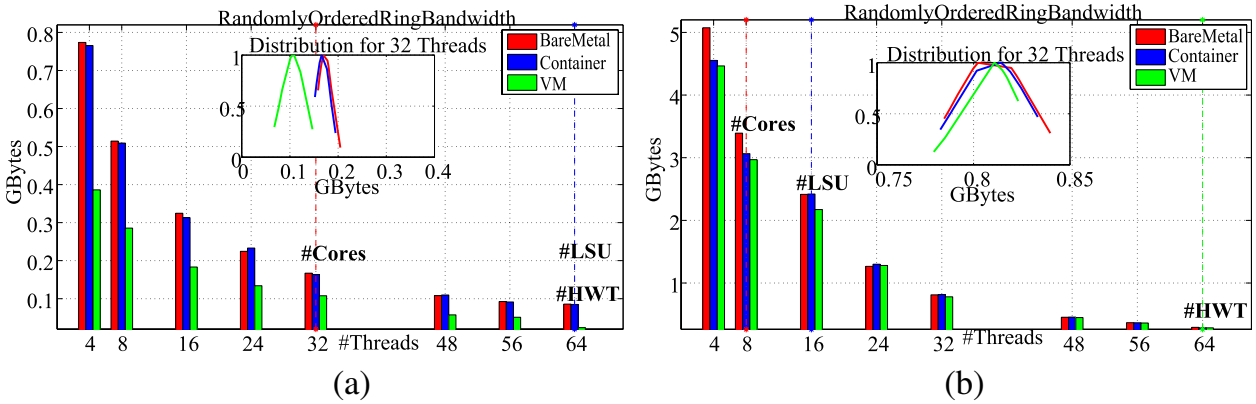

Fig. 17 Performance of RandomlyOrderedRingBandwidth on BareMetal-, Container-, and VM-based environments on (**a**) x86. (**b**) OpenPOWER

AvgPingPong bandwidth, similar behavior for all three environments is observed on the OpenPOWER system like random-order ring bandwidth. In x86 case, the performance boosts as the number of threads increases until the entire active bandwidth is exhausted, and beyond that, it starts to drop.

Inference: The more emphasis on random ring bandwidth reports the available inter-node communication bandwidth per MPI process because for the parallel applications, most MPI processes communicate with other MPI processes on other SMT nodes. In all the experiments, diminishing characteristics are observed as the number of threads increases. The possible reason behind this could be, in the case of a single node, all the communication happens to be within the same node, which leads to a higher network bandwidth. However, as the number of processes increases, more cores start communicating with each other through the router. The bandwidth delivered by the router is relatively lower than the PCIe-3 bus used in the OpenPOWER system, which degrades the performance of the RandomRing benchmark due to the inter-node communication.

The OpenPOWER-enabled system incorporates an RDMA-based (Remote Direct Access Memory) communication engine capable of an InfiniBand network card to hide the inherent overhead issues seen on the network bus, which usually has fewer communication threads than the default socket-based design. It provides high-performance and low-overhead network transfers by delegating the packet building and processing to the network chip. Additionally, to achieve good performance, POWER architecture-aware tuning can be exercised in advance/automatically, such as chunk size per RDMA operation, fine-grained communication thread binding, the number of handler threads for data transferring, RDMA buffer pool size, etc., based on the testbed [30]. Finally, we conclude that to achieve optimal performance for the communication benchmark; the significant factors are the efficient communication channel, an optimized MPI communication method, and message size.

Disk performance

IOzone: The widely used IOzone[3] benchmark is utilized to investigate the disk performance of BareMetal, container, and VM-based environments on x86 and OpenPOWER systems. It stresses and evaluates a variety of file operations such as read, write, re-read, re-write, read backwards, strided read, fread, fwrite, random read/write, pread/pwrite variants, aio_read, aio_write, and mmap on all the three environments. The benchmark runs with a maximum file size of 15 GB and a record capacity of up to 16 MB.

Figures 18 and 19 capture the performance of the IOzone benchmark related to different read and write operations for BareMetal, container, and VM-based environments on x86 and OpenPOWER systems. The minimal performance penalty connected to various I/O operations is observed in the Container-based environment as compared to the native performance on x86 as well as OpenPOWER systems. The VM-based environment encounters a significant performance penalty compared to the BareMetal-based environment on x86 and OpenPOWER systems because of the high overhead related to I/O and write operations inside the virtualized environment. The VM-based environment offers a noticeable difference in the performance as the file size increases due to the inefficiency of the virtualization driver to deal with the large file size. As the file size is increased, I/O performance degrades proportionally because the container scheduler tries to reorder I/O aggressively to avoid starvation as much as possible.

[3] http://www.iozone.org/

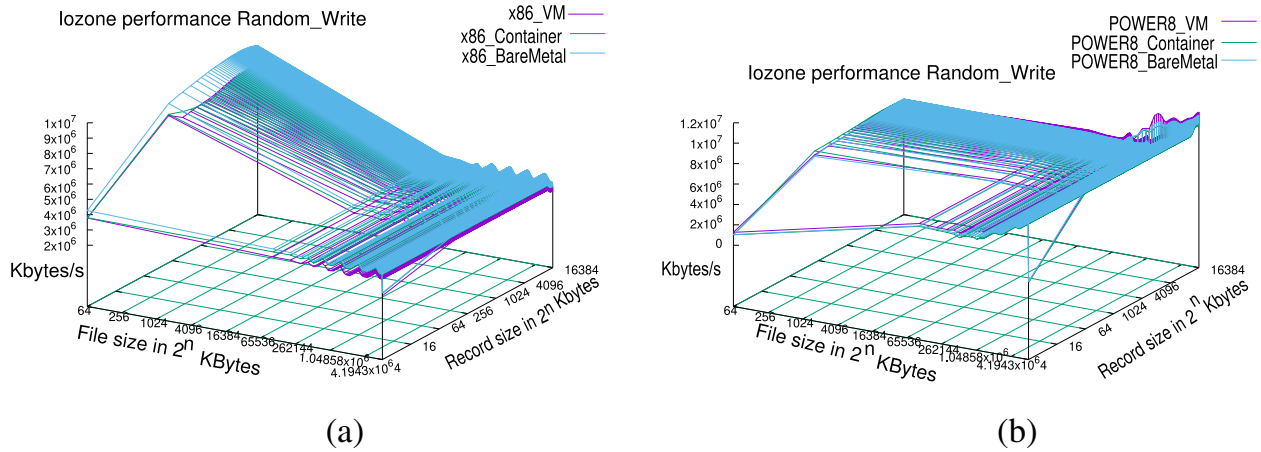

Fig. 18 Performance of IOzone Random_write (**a**) x86. (**b**) OpenPOWER

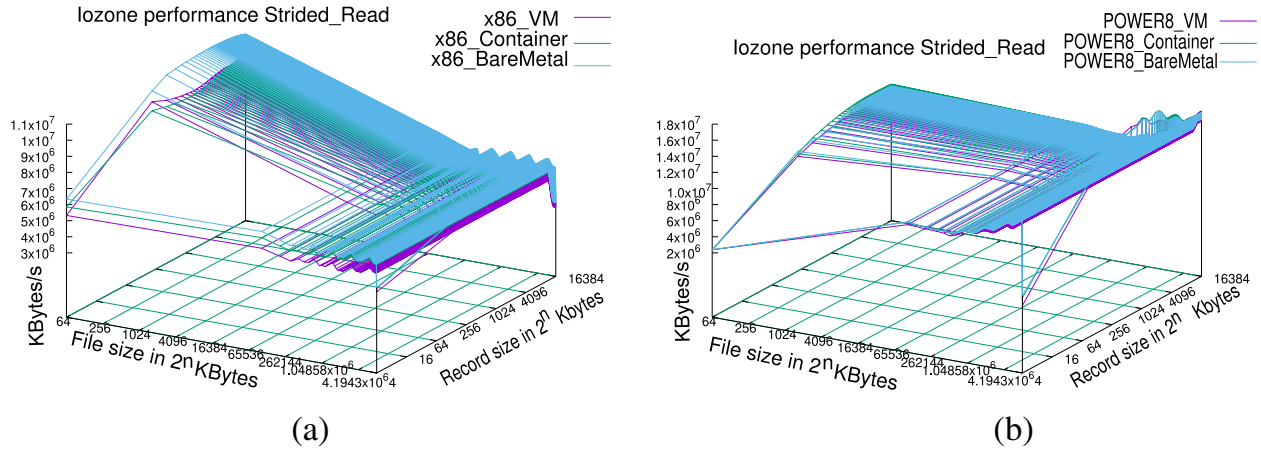

Fig. 19 Performance of IOzone Strided_Read (**a**) x86. (**b**) OpenPOWER

Summary & comparison

Here, the proposed work is summarized and compared with the existing contribution in this domain. Table 13 shows the maximal obtained results in all the experiments. Our experiments depict that container-based environments offer near-native performance in almost all aspects of the system compared to the BareMetal-based environment on both x86 and OpenPOWER systems. Although the x86-based environments present consistent and predictable performance behavior in all the experiments, but OpenPOWER-based environments show unpredictable nature in some cases. The tests display significant overhead in the case of a VM-based environment compared to the BareMetal-based environment on the x86 systems. However, the overhead is much less for VM-based environments using the OpenPOWER system. The inconsistent response is also observed in the case of HPL, MPIFFT, due to improper selection of some threads for all the three environments on the OpenPOWER system. The container-based environment outperforms the BareMetal-based environment in some benchmark experimentation; it may be because of the optimized library, and less interference involvement in the container environment. Moreover, an entirely different behavior is encountered in a few instances of our experiments. The reason behind this is not evident to us. A more detailed analysis for these corner cases can be done as a part of the future work. Table 14 shows that we stressed all the sub-components of the container-based HPC environments on OpenPOWER and x86 systems.

The state-of-the-art CPU and GPU implementation often suffer from limited performance while solving large-scale simulations. These applications are influenced by complex irregular access patterns and low arithmetic intensity. OpePOWER systems provide excellent benefits due to high data transfer memory bandwidth and

Table 13 Summary of the proposed work

Workload		x86			OpenPOWER		
		BareMetal	**Container**	**VM**	**BareMetal**	**Container**	**VM**
HPL (GFLOPS)		783.88[± 2.16]	782.12[± 2.15]	619.32[± 3.15]	213.54[± 1.72]	212.95[± 1.86]	203.22[± 2.06]
DGEMM	Single (GFLOPS)	394.44[± 1.76]	393.53[± 1.42]	315.49[± 1.99]	215.18[± 2.67]	213.65[± 2.61]	205.48[± 2.06]
	Star (GFLOPS)	232.56[± 0.86]	232.52[± 0.92]	216.49[± 1.96]	024.89[± 2.01]	024.21[± 2.03]	022.25[± 1.68]
FFT	Star (GFLOPS)	02.68[± 0.31]	02.62[± 0.25]	02.13[± 0.32]	01.48[± 0.37]	01.47[± 0.34]	01.46[± 0.33]
	MPI (GFLOPS)	10.82[± 0.43]	10.82[± 0.48]	07.05[± 0.58]	07.91[± 0.27]	07.87[± 0.42]	07.74[± 0.36]
Stream	Single(Triad) (GFLOPS)	13.14[± 0.74]	13.15[± 0.70]	11.66[± 0.81]	25.50[± 2.70]	24.94[± 2.75]	24.85[± 2.13]
	Star(Triad) (GFLOPS)	11.26[± 1.11]	11.17[± 1.02]	10.80[± 0.44]	10.69[± 0.53]	09.98[± 0.68]	09.32[± 1.23]
RandomAccess	Star (GUPs)	0.053[± 0.04]	0.052[± 0.03]	0.020[± 0.03]	1.60[± 0.03]	1.58[± 0.03]	1.49[± 0.05]
	MPI (GUPs)	0.038[± 0.04]	0.028[± 0.03]	0.019[± 0.04]	0.022[± 0.04]	0.022[± 0.04]	0.021[± 0.04]
PTRANS (GBs)		4.19[± 0.14]	4.20[± 0.18]	0.79[± 0.30]	4.59[± 0.10]	4.58[± 0.10]	4.43[± 0.26]
b_eff	AvgPingPong Latency(usec)	14.06[± 0.98]	14.00[± 0.98]	33.23[± 1.57]	01.07[± 0.08]	01.07[± 0.08]	01.08[± 0.08]
	RandomlyOrderedRing Latency(usec)	024.084[± 0.57]	024.847[± 1.06]	145.035[± 4.14]	001.993[± 0.15]	002.011[± 0.17]	001.993[± 0.19]
	AvgPingPong Bandwidth(GBs)	04.053[± 0.17]	04.035[± 0.19]	03.858[± 0.30]	11.430[± 0.95]	10.517[± 1.04]	09.396[± 0.72]
	RandomlyOrderedRing Bandwidth(GBs)	00.774[± 0.26]	00.766[± 0.19]	00.386[± 0.22]	05.075[± 0.67]	04.553[± 0.78]	04.465[± 0.63]

Table 14 Sub-components focused by our work

			x86 and OpenPOWER
System Sub-components Stress	Compute		✓
	Memory		✓
	Interconnect	Bandwidth	✓
		Latency	✓
	I/O		✓

an advanced near-memory accelerator than the conventional system architecture. Our proposed containerized HPC environment offers reduced deployment time, shorter latency, and better scalability, which is highly suitable for different big data workloads, having multiple executors distributed in the lightweight application container. It also provides fine-grained resource provisioning to reflect the big-data workload characteristics better. On the other hand, the OpenPOWER system has been introduced to suit Big Data workloads, Machine learning/AI, cloud/Containers by providing more threads per core and with a considerably large cache and memory bandwidth than the other platforms. It presents a modular architecture for the system accelerator to eliminate inherent I/O overhead and latency, increasing throughput. OperPOWER system also introduces NVlink, which significantly reduces the communication bottleneck between the CPU and GPU.

Conclusion & future work

In this work, we design and implement the Container-based HPC environments that rely on simple Linux namespace features build on the top of a) OpenPOWER and b) x86 systems. These container systems are experimentally analyzed for the performance compared to BareMetal, and VM-based environments. The applicability of these container-based environments on HPC is evaluated stressing the different subcomponents of the HPC systems using various benchmarks, HPCC, and IOzone.

The experiments show that the VM-based HPC solutions are not optimized enough to be utilized for workloads on the supercomputing facilities. Whereas, it is observed that, the Container-based HPC environments developed on x86 and OpenPOWER systems can be treated as the most feasible solution to fulfill the user's customized dynamic environment requirements, in shared HPC clusters without sacrificing the raw performance of the system. The performance of the OpenPOWER system represented a viable alternative to the x86 system for the HPC solution. The future generation systems are motivated by energy efficiency constraints together with the increase in scale, complexity, and heterogeneity. From a future perspective, our focus would be on making our container-based HPC environment solution to be energy-efficient. We also indent to explore the scalability test with the most recent OpenPOWER system comparing with the counterpart.

Acknowledgements
We sincerely thank Dr. Anshul Arora and Dr. Ankur Gupta for their valuable suggestions.

Authors' contributions
Animesh Kuity designed the model. He performed simulation and wrote the paper. All authors reviewed and edited the manuscript.

Funding
No funds have been received from any agency for this research.

Availability of data and materials
The data used to support the finding of this study are available from the corresponding author upon request.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 19 August 2021 Accepted: 8 November 2023
Published online: 15 December 2023

References

1. Jackson KR, Ramakrishnan L, Muriki K, Canon S, Cholia S, Shalf J, Wasserman HJ, Wright NJ (2010) Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference. IEEE, Indianapolis, p 159-168
2. Babu SA, Hareesh MJ, Martin JP, Cherian S, Sastri Y (2014) System Performance Evaluation of Para Virtualization, Container Virtualization, and Full Virtualization Using Xen, OpenVZ, and XenServer. In: 2014 Fourth International Conference on Advances in Computing and Communications. IEEE, Cochin, p 247-250. https://doi.org/10.1109/ICACC.2014.66
3. Younge AJ, Henschel R, Brown JT, von Laszewski G, Qiu J, Fox GC (2011) Analysis of Virtualization Technologies for High Performance Computing Environments. In: 2011 IEEE 4th International Conference on Cloud Computing. IEEE, Washington, DC, p 9-16. https://doi.org/10.1109/CLOUD.2011.29
4. Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L (2007) Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. ACM SIGOPS Oper Syst Rev 41(3):275–287
5. Beserra D, Moreno ED, Endo PT, Barreto J, Sadok D, Fernandes S (2015) Performance Analysis of LXC for HPC Environments. In: 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems. IEEE, Santa Catarina, p 358-363. https://doi.org/10.1109/CISIS.2015.53
6. Felter W, Ferreira A, Rajamony R, Rubio J (2015) An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, Philadelphia, p 171-172. https://doi.org/10.1109/ISPASS.2015.7095802
7. Xavier MG, Neves MV, Rossi FD, Ferreto TC, Lange T, De Rose CAF (2013) Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, Belfast, p 233-240. https://doi.org/10.1109/PDP.2013.41
8. Arango C, Dernat R, Sanabria J (2017) Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, abs/1709.10140. arXiv, Cornell, p 233-240
9. Li Z, Kihl M, Lu Q, Andersson JA (2017) Performance Overhead Comparison between Hypervisor and Container Based Virtualization. In: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA). IEEE, Taipei, p 955-962. https://doi.org/10.1109/AINA.2017.79
10. Kuity A, Peddoju SK (2017) Performance evaluation of container-based high performance computing ecosystem using openpower. In: Kunkel JM, Yokota R, Taufer M, Shalf J (ed) High Performance Computing. Springer International Publishing, Cham, Frankfurt, pp 290-308
11. Lu K, Chi W, Liu Y, Tang H (2009) HPVZ: A high performance virtual computing environment for super computers. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol 5737 LNCS, p 205-219
12. Khorassani KS, Chu CH, Subramoni H, Panda DK (2019) Performance evaluation of MPI libraries on GPU-enabled OpenPOWER architectures: Early experiences. In: International Conference on High Performance Computing. Springer, p 361-378
13. Ahmad WA, Bartolini A, Beneventi F, Benini L, Borghesi A, Cicala M, Forestieri P, Gianfreda C, Gregori D, Libri A, et al (2017) Design of an energy aware petaflops class high performance cluster based on power architecture. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, p 964-973
14. Matthews JN, Hu W, Hapuarachchi M, Deshane T, Dimatos D, Hamilton G, McCabe M, Owens J (2007) Quantifying the Performance Isolation Properties of Virtualization Systems. Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS 2007), ACM, San Diego, p 6-es
15. Scheepers MJ (2014) Virtualization and containerization of application infrastructure: A comparison. 21st twente student conference on IT, vol 21, Thijs Scheepers, p 1-7
16. Okafor K, Ononiwu G, Goundar S, Chijindu V, Udeze C (2021) Towards complex dynamic fog network orchestration using embedded neural switch. Int J Comput Appl 43(2):91–108
17. Zhang J, Lu X, Arnold M, Panda DK (2015) MVAPICH2 over OpenStack with SR-IOV: An efficient approach to build HPC clouds. In: Proceedings 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE, Shenzhen, p 71-80. https://doi.org/10.1109/CCGrid.2015.166
18. Mavridis I, Karatza H (2019) Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. Futur Gener Comput Syst 94:674–696
19. Adinetz AV et al (2015) Performance evaluation of scientific applications on POWER8. In: Jarvis S, Wright S, Hammond S (eds) High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation. PMBS 2014. Lecture Notes in Computer Science, vol 8966. Springer, Cham, p 24-45
20. Lu X, Shi H, Shankar D, Panda DK (2018) Performance characterization and acceleration of big data workloads on OpenPOWER system. In: 2017 IEEE International Conference on Big Data (Big Data), IEEE, Boston, p 213-222. https://doi.org/10.1109/BigData.2017.8257929
21. Reguly IZ, Keita AK, Zurob R, Giles MB (2016) High Performance Computing on the IBM Power8 Platform. In Taufer M., Mohr B., Kunkel J (eds) High Performance Computing. ISC High Performance 2016. Lecture Notes in Computer Science, vol 9945. Springer, Cham, p 235-254
22. Corporation I (2016) An Approach for Designing HPC Systems with Better Balance and Performance (1):1–7
23. Stillwell M, Schanzenbach D, Vivien F, Casanova H (2010) Resource allocation algorithms for virtualized service hosting platforms. J Parallel Distrib Comput 70(9):962–974
24. Kuity A, Peddoju SK (2023) CHPCe: Data Locality and Memory Bandwidth Contention-Aware Containerized HPC. In: Proceedings of the 24th International Conference on Distributed Computing and Networking. Association for Computing Machinery, New York, p 160–166. https://doi.org/10.1145/3571306.3571402
25. Dongarra JJ, Luszczek P (2006) Overview of the HPC Challenge Benchmark Suite. In: Proceeding of SPEC Benchmark Workshop. Citeseer
26. Okafor KC, Achumba IE, Chukwudebe GA, Ononiwu GC (2017) Leveraging fog computing for scalable IoT datacenter using spine-leaf network topology. J Electr Comput Eng 2017:11

27. Valero-Lara P, Martinez-Perez I, Mateo S, Sirvent R, Beltran V, Martorell X, Labarta J (2018) Variable batched DGEMM. In: 2018 26th Euromicro International Conference on Parallel. Distributed and Network-based Processing (PDP), IEEE, pp 363–367
28. Czechowski K, Battaglino C, McClanahan C, Iyer K, Yeung PK, Vuduc R (2012) On the communication complexity of 3D FFTs and Its Implications for Exascale. Association for Computing Machinery, New York. p 205–214. https://doi.org/10.1145/2304576.2304604
29. Murphy RC, Kogge PM (2007) On the memory access patterns of supercomputer applications: Benchmark selection and its implications. IEEE Transactions on Computers 56(7):937-45
30. Lu X, Shi H, Shankar D, Panda KDK (2017) Performance characterization and acceleration of big data workloads on OpenPOWER system. In: 2017 IEEE International Conference on Big Data (Big Data). IEEE, p 213-222

Publisher's Note

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

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com