Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Computer Architecture PhD program

Doctoral Thesis

Convergence of High Performance Computing, Big Data, and Machine Learning Applications on Containerized Infrastructures

Author: **Peini Liu**

Adviser: **Dr. Jordi Guitart**

Thesis advisors:
Dr. Jordi Guitart (Universitat Politècnica de Catalunya/Barcelona Supercomputing Center, Spain)

Barcelona, June 2023

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona
Computing Resources Orchestration and Management for AI Group
Barcelona Supercomupting Center
Emerging Technologies for Artificial Intelligence Group/Data-Centric Computing Group
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6-E201
08034, Barcelona, Spain

First printing, June 2023

"All men have the stars,
but they are not the same things for different people.
For some, who are travelers, the stars are guides.
For others they are no more than little lights in the sky.
For others, who are scholars, they are problems...
But all these stars are silent.
You – you alone will have the stars as no one else has them..."

The Little Prince
Antoine de Saint-Exupéry

Acknowledgements

As I stand on the threshold of a new chapter in my life, I am humbled and grateful for the countless individuals and organizations that have contributed to the realization of my dream. Completing a doctoral thesis is always not a solitary endeavor, and I have been fortunate enough to be surrounded by a community of extraordinary people who have lifted me up, inspired me, and helped me to achieve my goals.

I am honored to have written this thesis under the guidance of Dr. Jordi Guitart. As my supervisor throughout my doctoral career, Jordi has been a constant source of inspiration and guidance, imparting his wisdom and personality charm to me. I vividly recall our initial meeting, Jordi showed me some scripts and explained them in great detail, outlining their objectives and functionalities. Although I still can not tell how much I had understood that day, but I knew I had to learn to grow fast. In the next four years, Jordi provided me with meticulous guidance, ranging from identifying research topics, discussing research methods and results, to refining the content of academic papers. He introduced me to many research methodologies and listened to my infants' ideas. I am also grateful for his trust and sincerity toward me, which encouraged me to create work without fear. Jordi's unwavering commitment to excellence is reflected in his meticulous reading and invaluable feedback on my work, which has been crucial in shaping my way to research. His demand for excellence in my paper revisions has been a constant reminder of the importance of rigor and attention to detail in academic work. On the other hand, Jordi provided me with ample opportunities to present my research work and receive constructive feedback from peers. He encouraged me to participate in high-level academic and industrial conferences, broadening my academic horizons and expanding my knowledge in related fields. I am eternally grateful to him, as both a teacher and a friend, for seeing me through this journey, from start to end.

My gratitude goes beyond our remarkable teams and all collaborators. I am profoundly thankful to Guesseppe, as a great partner and friend. We had worked together through all my journey, and sometimes he had listened to my rambles and responded with full of care. I thank Ajay and David from Lenovo, who provided me with a powerful testbed and gave feedback on my work from an industry perspective. Last but not least, I gratefully thank Prof. Amir Taherkordi for his kindness in hosting me as a visitor at University of Oslo and for his willingness to give me his time so generously and unhesitatingly to discuss our research.

I am grateful for the time and effort of my internal reviewers, Dr. Raűl Sirvent, Dr. Juan José Costa, and Dr. Josep Lluís Berral, and external reviewers, Prof. Karim Djemame and Prof. Jean-Marc Pierson, who have dedicated themselves to reviewing my thesis and providing me with constructive feedback. Their comments and suggestions have been invaluable in refining the structure and content of my work.

I would like to express my sincere to the staff at BSC and UPC. Without the help of Xavier and Joana, the PhD program coordinator and secretary from UPC, I would not have been able to complete the program. I would also thank Katia from BSC-PMO office, who was always there helping me with my grant application, grant renovation and outcome submission. Antonio from BSC-HelpDesk helped me with my computer requests. Peter from Lenovo helped me with some settings on the cluster.

I am extremely thankful for the greatest friends that I have had met in Barcelona. To Chenle, Felippe and Guesseppe, we spent countless weekends together, discussing current progress and future, enjoying parks, cities and life. I will always remember these good days. To my badminton group friends, Junhao, Jiaqi, Jiaqui, Kaiqiang, Binlan, Chih-Chung, Carlas, Abel and Bernat, their presence in the gym brought joy to me after school life.

Finally, I would like to dedicate this dissertation to my beloved family. My parents' unconditional love and support have been the foundation of my courage and perseverance to pursue what enriches my life. Special thanks to Jie, who has been together with me to taste the delight and excitement and face the solitude and frustration of research. I wish you to continue pursuing your academic and professional goals and becoming a professor.

This dissertation, this degree, is for all of us.

Abstract

The convergence of High Performance Computing (HPC), Big Data (BD), and Machine Learning (ML) in the computing continuum is being pursued in earnest across the academic and industry. It is important to use a holistic approach that involves a multi-disciplinary team of experts in HPC, BD, and ML, and to use a combination of technologies and approaches. We envision virtualization and containerization technologies can be the basis for the convergence, because they reside as bridges between applications and infrastructures and provide well-known advantages, such as the encapsulation of specific software environments, which allows for customization, portability and reproducibility; the isolation of users from the underlying system and from other users, which allows for security and fault protection; and the agile and fine-grain resource allocation and balancing, which allows for efficient cluster utilization and failure recovery. However, challenges remain for this convergence at the containerization level due to the diversity of applications and hardware heterogeneity.

The challenges to be addressed are from different layers: in the infrastructure layer, virtualization/containerization must feature complete isolation of the applications in a multi-tenant environment, seamlessly and efficiently provide different resources (*e.g.,* GPUs, Infiniband, NUMA, etc.), allow agile and fine-grain dynamic resource provisioning to orchestrate resource sharing in those environments, also integrate HPC (*e.g.,* gang scheduling, affinity, preemption, topology-awareness, checkpoint/restore) and Cloud (*e.g.,* autoscaling, elasticity, migration) scheduling and resource management techniques, while providing fault tolerance, energy efficiency, and scalability. Moreover, in the platform layer, virtualization/containerization must fulfill applications requirements of portability and reproducibility by allowing the definition of encapsulated and customized diverse software stacks, and the efficient creation/termination of those software environments on demand. The general research question of this thesis is: **How to leverage virtualization/containerization in both the infrastructure layer and the platform layer to support efficiently the convergence of HPC, BD, and ML applications while taking advantage of heterogeneous HPC and Cloud resources?**

To answer that question, firstly, **we enable deployments of HPC, BD, and ML applications using containers** that allow the definition of encapsulated and customized software stacks for each application and provide seamless and efficient access to different resources through different configurations. This is the basis of the convergence.

Secondly, the challenge is to understand the performance impact of the various configurations and deployment options when using containers. Thus, **we perform several detailed performance analyses of containerization deployment options for diverse containerized applications on different hardware**. These performance analyses consider different containerization-level configurations, such as containerization technologies, granularities, affinities, and network interconnects. The obtained performance insights can be guidelines to derive placement policies when deploying applications to better utilize resources and achieve better application performance.

Thirdly, we enable DevOps for developing, building, and deploying these containerized applications and managing containers in multi-tenant, dynamic context circumstances by **establishing a platform to help the users to develop faster their containerized applications, to enable the deployment options analyzed before to feature**

efficient deployments, and to bring autonomic computing for continuously managing applications. This platform, so-called Scanflow-K8s, features a multi-agent multi-layered architecture (*i.e.,* application layer and infrastructure layer) that enables the online supervision of the end-to-end life-cycle of ML workflows on Kubernetes, as well as the deployment of containerized HPC workloads (through the Scanflow(MPI) package).

Finally, we leverage the knowledge learned from the performance analysis of the configuration and deployment of containerized HPC, BD, and ML workloads, and the availability of the autonomic management platform Scanflow-K8s, **to conduct autonomic management policies to schedule or manage various applications on the Scanflow-K8s platform**. On the one hand, we implement policies in the agent to evaluate the autonomic management and online supervision of the end-to-end life-cycle of ML workflows. On the other hand, we propose fine-grained scheduling policies for containerized HPC workloads in Kubernetes clusters, including decision-making in the granularity-aware agent in the application layer, and a MPI-aware plugin and a container-based task-group scheduling scheme for the Kubernetes Volcano scheduler in the infrastructure layer.

Resumen

La convergencia de Computación de Alto Rendimiento (HPC), Grandes Datos (BD) y Aprendizaje Automático (ML) en el continuum informático se está persiguiendo con seriedad en el ámbito académico e industrial. Es importante utilizar un enfoque holístico que involucre a un equipo multidisciplinario de expertos en HPC, BD y ML, y utilizar una combinación de tecnologías y enfoques. Prevemos que las tecnologías de virtualización y contenedorización pueden ser la base para la convergencia, porque residen como puentes entre las aplicaciones y las infraestructuras y proporcionan ventajas bien conocidas, como la encapsulación de entornos de software específicos, lo que permite la personalización, la portabilidad y la reproducibilidad; el aislamiento de los usuarios del sistema subyacente y de otros usuarios, lo que permite la seguridad y la protección contra fallos; y la asignación y el balanceo de recursos ágiles y detallados, lo que permite una utilización eficiente del clúster y la recuperación de fallos. Sin embargo, existen desafíos para esta convergencia a nivel de contenedorización debido a la diversidad de aplicaciones y la heterogeneidad del hardware.

Los desafíos que deben abordarse son de diferentes capas: en la capa de infraestructura, la virtualización/contenedorización debe presentar un aislamiento completo de las aplicaciones en un entorno de múltiples inquilinos, proporcionar de manera fluida y eficiente recursos diferentes (*e.g.,* GPU, Infiniband, NUMA, etc.), permitir la asignación dinámica de recursos ágil y detallada para orquestar el uso compartido de recursos en esos entornos, también integrar HPC (*e.g.,* planificación grupal, afinidad, preferencia, reconocimiento de topología, punto de control/restauración) y Cloud (*e.g.,* escalado automático, elasticidad, migración) planificación y técnicas de gestión de recursos, al mismo tiempo que proporciona tolerancia a fallos, eficiencia energética y escalabilidad. Además, en la capa de la plataforma, la virtualización/contenedorización debe cumplir con los requisitos de portabilidad y reproducibilidad de las aplicaciones al permitir la definición de diversas pilas de software encapsuladas y personalizadas, y la creación/terminación eficiente de esos entornos de software bajo demanda. La pregunta de investigación general de esta tesis es: **¿Cómo aprovechar la virtualización/contenerización tanto en la capa de infraestructura como en la capa de plataforma para respaldar de manera eficiente la convergencia de aplicaciones de HPC, BD y ML mientras se aprovechan los recursos heterogéneos de HPC y Cloud?**

Para responder a esa pregunta, en primer lugar, **habilitamos implementaciones de aplicaciones HPC, BD y ML utilizando contenedores** que permiten la definición de pilas de software encapsuladas y personalizadas para cada aplicación y proporcionan acceso fluido y eficiente a diferentes recursos a través de diferentes configuraciones. Esta es la base de la convergencia.

En segundo lugar, el desafío consiste en comprender el impacto en el rendimiento de las diversas configuraciones y opciones de implementación al utilizar contenedores. Por lo tanto, **realizamos varios análisis detallados de rendimiento de opciones de implementación de contenedores para diversas aplicaciones en diferentes hardware**. Estos análisis de rendimiento consideran diferentes configuraciones a nivel de contenedores, por ejemplo, tecnologías de contenedores, granularidades, afinidades e interconexiones de red. Los conocimientos de rendimiento obtenidos pueden servir como

directrices para derivar políticas de ubicación al implementar aplicaciones para utilizar mejor los recursos y lograr un mejor rendimiento de las aplicaciones.

En tercer lugar, habilitamos DevOps para desarrollar, construir e implementar estas aplicaciones en contenedores y gestionar los contenedores en contextos dinámicos y de múltiples usuarios **estableciendo una plataforma que ayude a los usuarios a desarrollar sus aplicaciones en contenedores más rápidamente, permita las opciones de implementación analizadas anteriormente para contar con implementaciones eficientes, y proporcione computación autónoma para gestionar continuamente las aplicaciones**. Esta plataforma, llamada Scanflow-K8s, cuenta con una arquitectura multiagente y de múltiples capas (*i.e.,* capa de aplicación y capa de infraestructura) que permite la supervisión en línea del ciclo de vida de extremo a extremo de los flujos de trabajo de ML en Kubernetes, así como la implementación de cargas de trabajo HPC en contenedores (a través del paquete Scanflow(MPI)).

Finalmente, aprovechamos el conocimiento aprendido del análisis de desempeño en la configuración e implementación de cargas de trabajo de HPC, BD y ML contenedorizadas, y la disponibilidad de una plataforma usable de administración autónoma (Scanflow-K8s), **para conducir políticas de gestión autónomas para programar o administrar varias aplicaciones en la plataforma Scanflow-K8s**. Por un lado, implementamos políticas en el agente para evaluar la gestión autónoma y la supervisión en línea del ciclo de vida de extremo a extremo de los flujos de trabajo de ML en Scanflow-K8s. Por otro lado, proponemos políticas de planificación detalladas para cargas de trabajo de HPC contenedorizadas en clústeres de Kubernetes, que incluyen en el agente una toma de decisiones consciente de la granularidad en la capa de la aplicación, y un complemento consciente con MPI y un esquema de planificación de grupos de tareas basado en contenedores para el planificador de Kubernetes Volcano en la capa de infraestructura.

Resum

La convergència de Computació d'Alt Rendiment (HPC), Grans Dades (BD) i Aprenentatge Automàtic (ML) al continuum informàtic s'està perseguint amb serietat a l'àmbit acadèmic i industrial. És important utilitzar un enfocament holístic que involucri un equip multidisciplinari d'experts en HPC, BD i ML, i utilitzar una combinació de tecnologies i enfocaments. Preveiem que les tecnologies de virtualització i contenidorització poden ser la base per a la convergència, perquè resideixen com a ponts entre les aplicacions i les infraestructures i proporcionen avantatges ben coneguts, com l'encapsulació d'entorns de programari específic, cosa que permet la personalització, la portabilitat i la reproductibilitat; l'aïllament dels usuaris del sistema subjacent i d'altres usuaris, cosa que permet la seguretat i la protecció contra fallades; i l'assignació i el balanceig de recursos àgils i detallats, la qual cosa permet una utilització eficient del clúster i la recuperació de fallades. No obstant això, hi ha desafiaments per a aquesta convergència a nivell de contenidorització a causa de la diversitat d'aplicacions i l'heterogeneïtat del maquinari.

Els desafiaments que cal abordar són de diferents capes: a la capa d'infraestructura, la virtualització/contenidorització ha de presentar un aïllament complet de les aplicacions en un entorn de múltiples inquilins, proporcionar de manera fluida i eficient recursos diferents (*e.g.,* GPU, Infiniband, NUMA, etc.), permetre l'assignació dinàmica de recursos àgil i detallada per orquestrar l'ús compartit de recursos en aquests entorns, també integrar HPC (*e.g.,* planificació grupal, afinitat, preferència, reconeixement de topologia, punt de control/restauració) i Cloud (*e.g.,* escalat automàtic, elasticitat, migració) planificació i tècniques de gestió de recursos, alhora que proporciona tolerància a fallades, eficiència energètica i escalabilitat. A més, a la capa de la plataforma, la virtualització/contenidorització ha de complir amb els requisits de portabilitat i reproductibilitat de les aplicacions en permetre la definició de diverses piles de programari encapsulades i personalitzades, i la creació/terminació eficient d'aquests entorns de programari sota demanda. La pregunta d'investigació general d'aquesta tesi és: **Com aprofitar la virtualització/contenidorització tant a la capa d'infraestructura com a la capa de plataforma per recolzar de manera eficient a la convergència d'aplicacions HPC, BD i ML mentre s'aprofiten els recursos heterogenis de HPC i Cloud?**

Per respondre a aquesta pregunta, en primer lloc, **habilitem implementacions d'aplicacions HPC, BD i ML utilitzant contenidors** que permeten la definició de piles de programari encapsulades i personalitzades per cada aplicació i proporcionen un accés eficient i constant a diferents recursos a través de diferents configuracions. Aquesta és la base de la convergència.

En segon lloc, el repte consisteix a comprendre l'impacte en el rendiment de les diferents configuracions i opcions de desplegament en utilitzar contenidors. Per tant, **realitzem diverses anàlisis detallades de rendiment de les opcions de desplegament de contenidors per a diverses aplicacions en contenidors en diferent maquinari**. Aquestes anàlisis de rendiment tenen en compte diferents configuracions a nivell de contenidors, com ara tecnologies de contenidorització, granularitats, afinitats i interconnexions de xarxa. Els coneixements de rendiment obtinguts poden ser pautes per

derivar polítiques de col·locació en desplegar aplicacions per utilitzar millor els recursos i aconseguir un millor rendiment de l'aplicació.

En tercer lloc, habilitem DevOps per desenvolupar, construir i desplegar aquestes aplicacions en contenidors i gestionar els contenidors en contextos dinàmics i de múltiples usuaris **establint una plataforma per ajudar els usuaris a desenvolupar més ràpidament les seves aplicacions en contenidors, per habilitar les opcions de desplegament analitzades anteriorment per comptar amb desplegaments eficients, i per portar la informàtica autònoma per a la gestió contínua d'aplicacions**. Aquesta plataforma, anomenada Scanflow-K8s, presenta una arquitectura multi-agent multicapa (*i.e.,* capa d'aplicació i capa d'infraestructura) que permet la supervisió en línia del cicle de vida d'extrem a extrem dels fluxos de treball de ML a Kubernetes, així com la implementació de càrregues de treball HPC en contenidors (a través del paquet Scanflow (MPI)).

Finalment, aprofitem el coneixement aprés de l'anàlisi de rendiment en la configuració i implementació de càrregues de treball d'HPC, BD i ML contenidoritzades, i la disponibilitat d'una plataforma usable d'administració autònoma (Scanflow-K8s), **per conduir polítiques de gestió autònomes per programar o administrar diverses aplicacions a la plataforma Scanflow-K8s**. D'una banda, implementem polítiques a l'agent per avaluar la gestió autònoma i la supervisió en línia del cicle de vida d'extrem a extrem dels fluxos de treball de ML a Scanflow-K8s. D'altra banda, proposem polítiques de planificació detallades per a càrregues de treball d'HPC contenidoritzades en clústers de Kubernetes, que inclouen a l'agent una presa de decisions conscient de la granularitat a la capa de l'aplicació, i un complement conscient amb MPI i un esquema de planificació de grups de tasques basat en contenidors per al planificador de Kubernetes Volcano a la capa d'infraestructura.

Contents

List of Figures

List of Tables

Listings

Nomenclature

AI	Artificial Intelligence
AIOps	Artificial Intelligence for IT Operations
BD	Big Data
BSC	Barcelona Supercomputing Center
CNN	Convolutional Neural Network
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CRD	Custom Resource Definition
DevOps	Software Development and IT Operations
DNN	Deep Neural Network
DL	Deep Learning
GPU	Graphic Processing Unit
GPFS	General Parallel File System
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
HPDA	High Performance Data Analysis
IPC	Inter-Process Communication
IPoIB	IP over InfiniBand
IT	Information Technology
ICT	Information and Computing Technology
KVM	Kernel-based Virtual Machine
K8s	Kubernetes
LXC	Linux Container
ML	Machine Learning
MLOps	Machine Learning Operations
MPI	Message Passing Interface
NAS	Numerical Aerodynamic Simulation
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access Architecture
OverlayFS	Overlay Filesystem
PID	Process Identifier
QP	Queue Pair
RDMA	Remote Direct Memory Access
RNN	Recurrent Neural Network
TPU	Tensor Processing Unit

UID	User Identifier
UMA	Uniform Memory Access
UPC	Universitat Politècnica de Catalunya
VM	Virtual Machine
YARN	Yet Another Resource Negotiator
2L	Two-level

Chapter 1

Introduction

In this chapter, the context and motivation of the thesis are presented in Section 1.1. Then, the objectives and contributions are introduced in Section 1.2. Section 1.3 explains the overview outline of the thesis.

1.1 Context and Motivation

The computing paradigm has been evolving over the last three decades (as shown in Figure 1.1). In the early 1975s, the traditional High Performance Computing (HPC) communities solved scientific problems by using modeling and simulation through supercomputers [18][131].

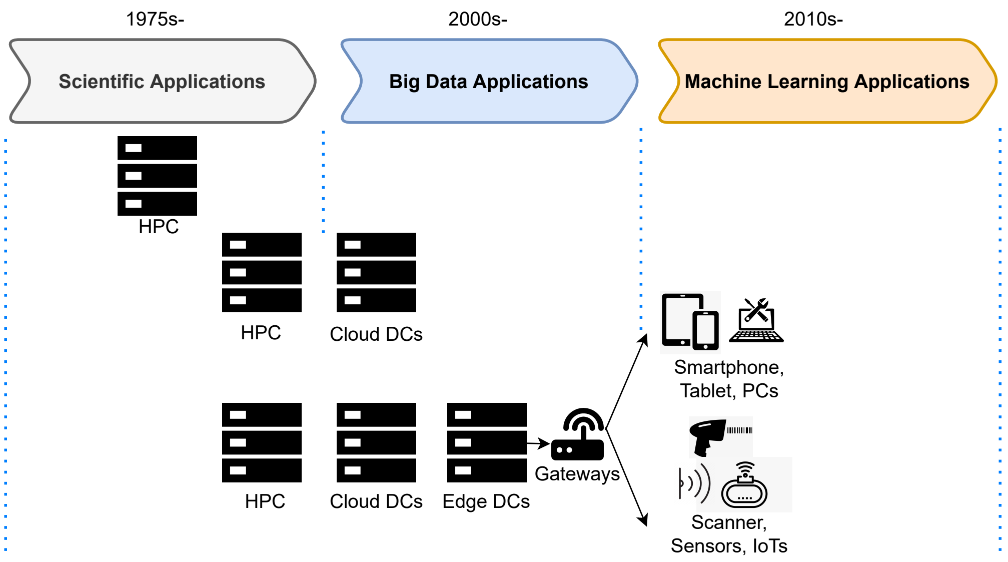


Figure 1.1: Computing evolving stages over the last three decades.

However, from the 2000s, Big Data (BD) started to change the way people understood and harnessed the power of data, both in the business and research domains [138]. Scientific computation problems had been faced with the need to analyze increasing amounts of data as part of their application workflows, and the science-based model was being combined with data-driven models to represent complex systems and phenomena [18][132]. Thus, HPC and BD started converging to meet large-scale data processing challenges to enable High Performance Data Analysis (HPDA) in HPC and the Cloud [9][22][139][40].

Currently, from the 2010s, Machine Learning (ML)-enabled science, engineering, arts, health, and business are now driving a multibillion-dollar industry and playing an increasingly important role in human society [137]. Innovative ML applications provide technological breakthroughs which have powered transformational solutions for BD challenges. The high-quality data (*e.g.,* ImageNet [156], Coco [95]), novel pattern recognition algorithms (*e.g.,* Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Deep Neural Network (DNN) [116]), accelerated computing (*e.g.,* Graphic Processing Unit (GPU) [183], Tensor Processing Unit (TPU) [195]), and open-source software platforms and frameworks (*e.g.,* Tensorflow [175], Pytorch [143]) lead to ML with good results in different tasks such as computer vision, machine translation, recommendation systems, and speech recognition [64][55][81][111][26]. However, the challenge remains for ML towards optimal exploitation of large-scale data and extreme-scale computing. Therefore, ML is joining the HPC and BD convergence. On the one hand, HPC platforms can be exploited to be used for ML model distributed training and inference. Furthermore, the mixed computing continuum architecture (HPC/Cloud/Edge) could power the ML world with a more solid origin of data and computation, and bring ML close to the end-user [152][151].

The convergence of HPC, BD, and ML is being pursued in earnest across the academic [18][9][64] and industry [34][133]. It is important to use a holistic approach that involves a multi-disciplinary team of experts in HPC, BD, and ML, and to use a combination of technologies and approaches. We envision virtualization and containerization technologies can be the basis towards the convergence, because they provide well-known advantages [202], such as the encapsulation of specific software environments, which allows for customization, portability and reproducibility [90]; the isolation of users from the underlying system and from other users, which allows for security and fault protection; and the agile and fine-grain resource allocation and balancing, which allows for efficient cluster utilization and failure recovery [37]. However, as shown in Figure 1.2, there are still some significant challenges for this convergence at containerization level due to the diversity of applications and the hardware heterogeneity. Thus, challenges have to be addressed from different layers: In the infrastructure layer, virtualization/containerization must feature complete isolation of the applications in a multi-tenant environment, seamlessly and efficiently provide different resources (*e.g.,* GPUs, Infiniband, NUMA, etc.), allow agile and fine-grain dynamic resource provisioning to orchestrate resource sharing in those environments, also integrate HPC (*e.g.,* gang scheduling, affinity, preemption, topology-awareness, checkpoint/restore) and Cloud (*e.g.,* autoscaling, elasticity, migration) scheduling and resource management techniques, while providing fault tolerance, energy efficiency, and scalability. Moreover, in the platform layer, virtualization/containerization must fulfill applications requirements of portability and reproducibility by allowing the definition of encapsulated and customized diverse software stacks, and the efficient creation/termination of those software environments on demand. In this context, the general research question that this thesis aims to answer is the following:

How to leverage virtualization/containerization in both the infrastructure layer and the platform layer to support efficiently the convergence of HPC, BD, and ML applications while taking advantage of heterogeneous HPC and Cloud resources?

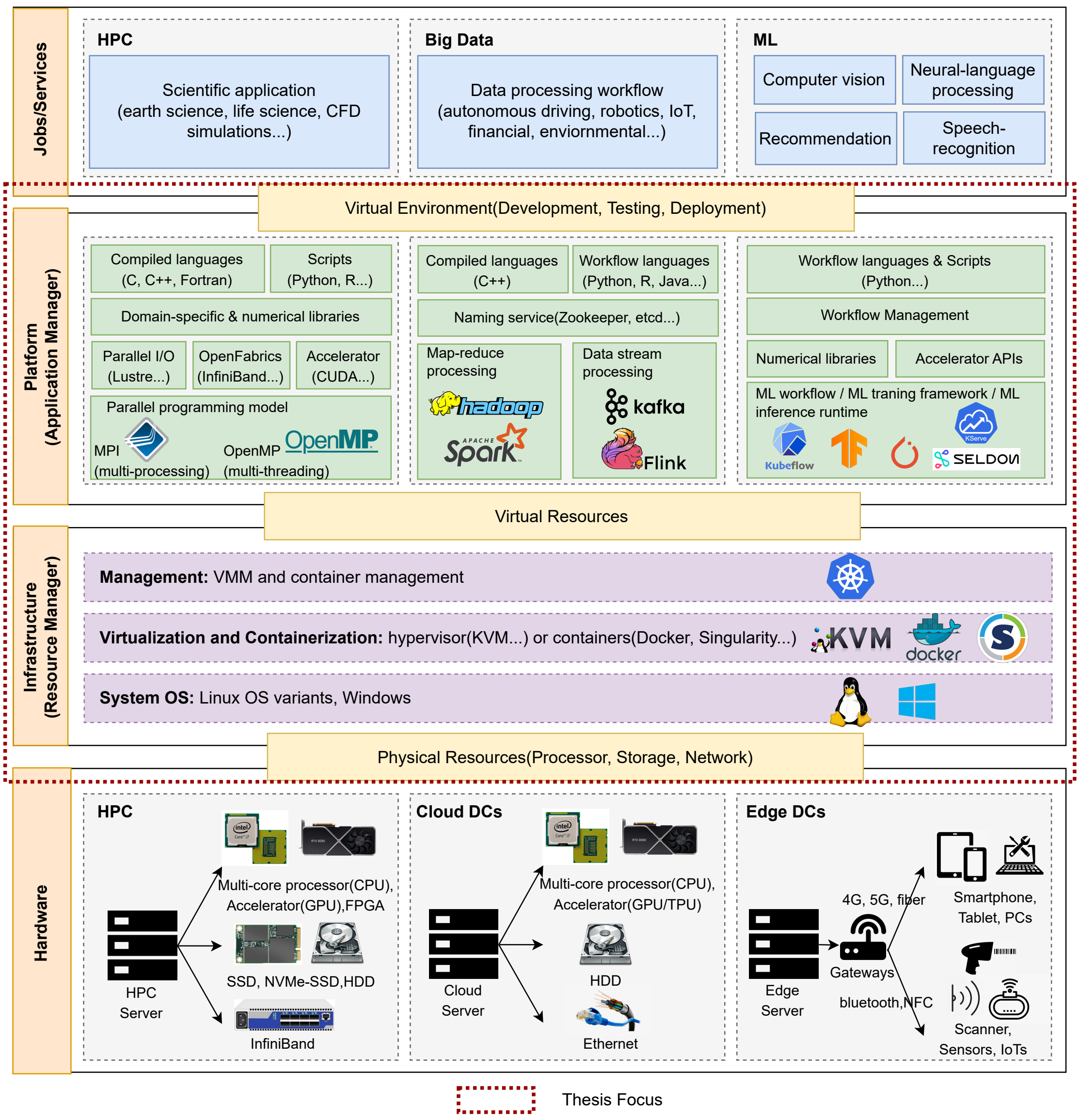

Jobs/Services
HPC
Scientific application (earth science, life science, CFD simulations...)
Big Data
Data processing workflow (autonomous driving, robotics, IoT, financial, enviornmental...)
ML
Computer vision
Neural-language processing
Recommendation
Speech-recognition
Virtual Environment(Development, Testing, Deployment)
Platform (Application Manager)
Compiled languages (C, C++, Fortran)
Scripts (Python, R...)
Domain-specific & numerical libraries
Parallel I/O (Lustre...)
OpenFabrics (InfiniBand...)
Accelerator (CUDA...)
Parallel programming model
MPI (multi-processing)
OpenMP (multi-threading)
Compiled languages (C++)
Workflow languages (Python, R, Java...)
Naming service(Zookeeper, etcd...)
Map-reduce processing
Data stream processing
Workflow languages & Scripts (Python...)
Workflow Management
Numerical libraries
Accelerator APIs
ML workflow / ML traning framework / ML inference runtime
Virtual Resources
Infrastructure (Resource Manager)
Management: VMM and container management
Virtualization and Containerization: hypervisor(KVM...) or containers(Docker, Singularity...)
System OS: Linux OS variants, Windows
Physical Resources(Processor, Storage, Network)
Hardware
HPC
HPC Server
Multi-core processor(CPU), Accelerator(GPU),FPGA
SSD, NVMe-SSD,HDD
InfiniBand
Cloud DCs
Cloud Server
Multi-core processor(CPU), Accelerator(GPU/TPU)
HDD
Ethernet
Edge DCs
Edge Server
4G, 5G, fiber
Gateways
bluetooth,NFC
Smartphone, Tablet, PCs
Scanner, Sensors, IoTs
Thesis Focus

Figure 1.2: Convergence of HPC, BD, and ML applications on containerized infrastructures.

1.1.1 Infrastructure Opportunities

BD/ML engineers are not experts in using emerging HPC hardware, and likewise, HPC engineers could not fully understand Cloud architectures. But all these heterogeneous resources with different characteristics can provide computing, network, and storage capabilities, making a huge contribution to improving application performance. For instance, HPC hardware has attempted to improve the performance of applications over computational accelerators (*e.g.,* GPU) [183], high-speed interconnection networks (*e.g.,* InfiniBand, Inter Omni-path) [108][76], and parallel file systems (*e.g.,* GPFS, Lustre) [146][145][69]. Therefore, there are some opportunities in the infrastructure layer regarding using containerization and container management to provide transparent, isolated, trustworthy, and efficient containerized infrastructures for a wide range of applications.

- **Containerization**: Both HPC, BD, and ML communities can adopt the virtualization/containerization technologies taking advantage of those virtual machines [73][29][147] and containers [70][200][17][209] for hardware and facility transparency, providing encapsulation of specific software environment with portability and reproducibility and establishing a secure and isolated execution environment. Therefore, studying containerization technologies (*e.g.,* Docker, Singularity, etc.) and deployment configurations (*e.g.,* number of containers, container interconnections, etc.) to leverage the features of different hardware to improve the performance of various different types of applications is important.

- **Resource Management**: All the HPC, BD, and ML ecosystems launching massive jobs/services on a large-scale system will require support to reduce jobs/services response time, monitor jobs/services in real-time, analyze the runtime jobs/services status, and manage and schedule jobs/services to better utilize the whole computation resources and migrate the jobs/services to face failures and achieve better performance. Running diverse HPC, BD, and ML applications with different characteristics on different resources is happening. Thus, a common container management tool, which integrates HPC (*e.g.,* gang scheduling, affinity, preemption, topology-awareness, checkpoint/restore) and Cloud (*e.g.,* autoscaling, elasticity, migration) scheduling and resource management techniques, is expected to solve these challenges (*e.g.,* fault tolerance, energy efficiency, and scalability).

1.1.2 Platform Opportunities

Platform opportunities come from the diversity of the applications and their corresponding software frameworks. HPC applications take advantage of parallel processing architectures to perform large-scale computations efficiently and handle large amounts of data effectively [58][130]. BD applications, typically executed as batch processing jobs, use MapReduce model to automate repetitive tasks by executing a large number of jobs in a non-interactive mode [45][5][194]. ML applications focus on learning models from data and making predictions by using the trained model. From a runtime perspective, the ML training and batch ML inference could be executed as offline jobs and may take days to complete, whereas the online ML inference service is realized as a long-run service that is able to deal with dynamic prediction queries from end-users [97][98][203][92]. These bring challenges in the platform layer for containers to consider and support the development, testing, deployment, and operation of wide range types of applications.

- **Frameworks**: HPC applications work well with HPC software and libraries, such as MPI [39], OpenMP, etc. BD applications are supported by reliable, scalable distributed computing libraries, such as, Hadoop [3], Spark [6], etc. ML applications are empowered by distributed frameworks, such as Tensorflow [175], Pytorch [143], etc. ML online inference services are served by Seldon [161], Tensorflow Serving [134][176], etc. To utilize these frameworks in a containerized environment, efforts should be provided on how to combine those frameworks with containers (*e.g.,* encapsulate applications into containers, configure frameworks inside containers). On the other hand, the application information shown or status gathered by the frameworks are valuable, thus can be used together with the infrastructure information to improve the deployment and operation efficiency.

- **Software development and IT operations**: Software Development and IT Operations (DevOps) is a popular software development methodology invented in the context of Cloud computing and containerization to improve the software development and delivery process. It is complementary to agile software development, emphasizes collaboration and communication between development and operations teams, and aims to automate and streamline the software delivery process [79]. Artificial Intelligence for IT Operations (AIOps) brings intelligence to DevOps. It combines Big Data and Machine Learning to automate IT operations processes [43]. AIOps platform enables the concurrent use of multiple data sources, data collection methods, and analytical and presentation technologies as it can help to improve the performance, availability and scalability of IT systems, and also by providing automated and intelligent ways to manage and operate the systems in a more efficient way [44]. DevOps teams can also improve the feedback loop and get more insights from the AIOps platform. Therefore, the opportunity for HPC, BD, and ML communities is to all adopt the DevOps cycle for the development and operation collaboration under the containerized environment, and leverage AIOps platform to enhance the IT operations in an automated and intelligent way to improve software development and delivery process.

1.2 Objectives and Contributions

The purpose of this thesis focuses on leveraging containerization technologies for the convergence of HPC, BD, and ML applications on containerized infrastructures. Considering the opportunities presented in Section 1.1, we divide our main purpose into four objectives.

1.2.1 Objective 1: Enable deployments of HPC, BD, and ML applications using containers

Containerization technology offers an alternative opportunity to operate and package the workloads without being limited by the performance degradation of using virtual machines (VMs). As the basis for their convergence, **our objective is to enable deployments of HPC, BD, and ML workloads using containers**, by allowing the definition of encapsulated and customized diverse software stacks for each application and providing seamless and efficient access to different HPC resources, so that **applications can make an efficient use of the containers (through different configura-**

tions) to improve their performance, mainly featuring the following container-level considerations:

- Containerization Technology: Different container runtime implementations (*e.g.,* Docker, Singularity, etc.).
- Container Granularity: Different number of containers per node.
- Container Affinity: Different CPU/Memory affinity configurations.
- Container Network: Different networking fabrics (*e.g.,* MACVLAN network or VxLAN network).

Our contributions to achieve this objective are as follows, and have resulted in publications [99][101][97]. BD applications have also been containerized in this manner, as published formerly by our group [160], hence, this is not included as a contribution in this thesis:

1. **O1-C1:** We propose the multi-container deployments (*i.e.,* partitioning the processes belonging to each application into different containers) for HPC applications, and derive corresponding affinity settings for each container belonging to the deployment scheme. We enable these configurations both in Docker and Singularity.
2. **O1-C2:** In addition to the above settings, we also enable different network interconnections for multiple containers belonging to an HPC application in an InfiniBand cluster.
3. **O1-C3:** We enable containerization for ML workflows using Docker, in both ML training stage and ML inference stage. Particularly, we adopt multi-container deployment schemes and affinity settings for online ML inference services.

1.2.2 Objective 2: Understand the performance of HPC, BD, and ML applications running on containers

After being able to deploy HPC, BD, and ML the applications on containers with different deployment options, the challenge is to choose for each application the configurations and deployment options that provide better performance when using containers. Therefore, **our objective is understanding the impact on the performance of different applications running on different platforms using various container-level deployment options**. The results could provide us the knowledge of what the bottleneck of deployment options is and **how to choose the most adequate deployment schemes for containerized applications to achieve the best performance**.

Our contributions to achieve this objective are as follows, and have resulted in publications [99][101][102]. We consider that most of our conclusions in the objective would also apply to BD workloads. Some related analyses of BD worklods with containers can be found in a former publication from our group [160]:

1. **O2-C1:** We perform a performance analysis of distinct multi-container deployment schemes for HPC workloads comprising i) different containerization technologies, ii) different container granularity, iii) different container processor and memory affinity configurations, iv) different hardware platform settings (*e.g.,* Non-Uniform Memory Access (NUMA), Uniform Memory Access (UMA)), v) different application subscription modes (exactly- or over-subscribed mode).

2. **O2-C2:** We perform a detailed performance characterization of different containerization technologies (including Docker and Singularity) for HPC workloads on InfiniBand clusters through different dimensions, namely network interconnects (including Ethernet and InfiniBand) and protocols (including TCP/IP and Remote Direct Memory Access (RDMA)), networking modes (including host, MACVLAN, and overlay networking), and processor and memory affinity.

3. **O2-C3:** We perform a performance characterization of multiple deployment schemes for online ML inference services that feature different degrees of container granularity and we set the corresponding distribution of application working threads and resources to each container to serve the model. In addition, we investigate CPU/Memory affinity for each container belonging to an online ML inference service as part of the former deployment schemes.

1.2.3 Objective 3: Devise an autonomic management platform for containerized HPC, BD, and ML applications

After being able to containerize various applications and understand the impact of the various deployment options, the challenge is how to enable DevOps to develop, build, and deploy these containerized applications efficiently and manage containers in multi-tenant and dynamic contexts. Therefore, **our objective is to devise a platform** (so-called Scanflow-K8s) to i) **help the users to develop faster their containerized applications and build images**; ii) **enable the deployment options analyzed before to feature efficient deployments**; iii) **bring autonomic computing for continuously managing applications**.

Our contributions to achieve this objective are as follows, and have resulted in publications [98][100]:

1. **O3-C1:** We enable an agent-based approach to leverage autonomic computing. This multi-agent system aims to maintain robustness and satisfy requirements at the application layer. We design the Scanflow multi-agent approach by using triggers, primitives, and strategies.

2. **O3-C2:** We investigate an architecture with abilities for two-layered management (*i.e.,* application layer and infrastructure layer). Based on the architecture, we establish a real platform Scanflow-K8s, a functional agent-based platform that enables autonomic management and online supervision of the end-to-end life-cycle of ML workflows on Kubernetes. Moreover, various teams could use Scanflow-K8s to build and deploy their ML workflows in different phases.

3. **O3-C3:** We extend the platform in the application layer with a Scanflow(MPI) package, which allows the users to use the Scanflow-client Python library to easily define and build HPC workloads locally and submit MPI jobs to Scanflow-server to be deployed in a Kubernetes cluster.

1.2.4 Objective 4: Optimize container management and scheduling for containerized HPC, BD, and ML applications

Utilizing the knowledge and the platform provided by Objective 2 and Objective 3, respectively, **our objective is to design autonomic management mechanisms and**

efficient scheduling policies that outperform the state-of-the-art by allowing agile and fine-grain dynamic resource provisioning to orchestrate resource sharing and integrating HPC and Cloud scheduling and resource management techniques. Therefore, on the one hand, we implement policies in the agent to feature the **autonomic management and online supervision of the end-to-end life-cycle of ML workflows** on Scanflow-K8s. On the other hand, we propose **fine-grained scheduling policies for containerized HPC workloads** in Kubernetes clusters (*i.e.,* Scanflow(MPI)-K8s platform).

Our contributions to achieve this objective are as follows, and have resulted in publications [98][100]:

1. **O4-C1:** We conduct experiments on Scanflow-K8s to illustrate the features of the agents and evaluate the feasibility and effectiveness of our agent-based approach for autonomic management of ML workflows. We define and implement policies for Scanflow agents to support autonomous management for ML workflows in each different dynamic context.

2. **O4-C2:** We propose fine-grained scheduling policies for containerized HPC workloads in Kubernetes clusters, focusing on multi-container deployments according to the application profile, using CPU/memory affinity and the idea of even distribution. We implement and adopt our scheduling schemes on a Scanflow(MPI)-K8s. We develop policies for the granularity-aware agent in the application layer, and the MPI-aware plugin and the container-based task-group scheduling scheme for the Kubernetes Volcano scheduler in the infrastructure layer.

Figure 1.3 represents these objectives and contributions and their relationships.

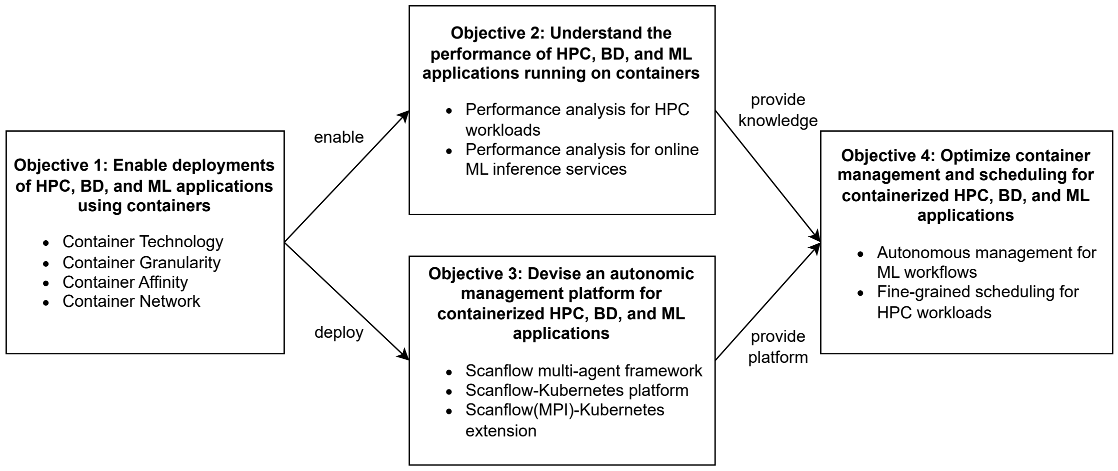

Objective 1: Enable deployments of HPC, BD, and ML applications using containers
• Container Technology
• Container Granularity
• Container Affinity
• Container Network
enable
deploy
Objective 2: Understand the performance of HPC, BD, and ML applications running on containers
• Performance analysis for HPC workloads
• Performance analysis for online ML inference services
Objective 3: Devise an autonomic management platform for containerized HPC, BD, and ML applications
• Scanflow multi-agent framework
• Scanflow-Kubernetes platform
• Scanflow(MPI)-Kubernetes extension
provide knowledge
provide platform
Objective 4: Optimize container management and scheduling for containerized HPC, BD, and ML applications
• Autonomous management for ML workflows
• Fine-grained scheduling for HPC workloads

Figure 1.3: Thesis objectives and contributions diagram.

Overall, these contributions demonstrate the feasibility and benefits of converging HPC, BD, and ML applications using containerization technology on containerized infrastructures, providing insights into performance analysis for different multi-container deployment options, conducting a platform for autonomous management for containerized applications, and presenting efficient container management and scheduling schemes.

1.3 Thesis Outline

The content of the thesis is organized in the following chapters and appendices:

- **Chapter 2** presents the literature review of HPC, BD, and ML applications and features, as well as state-of-the-art of containerization and container management methods and technologies.

- **Chapter 3** enables the multi-container deployments of HPC applications with various container-level configurations and supports different configurations for HPC applications or infrastructures. The first two cases present a systematic performance comparison and analysis of multi-container deployment schemes for HPC workloads on a single-node platform, which considers different containerization technologies (including Docker and Singularity), different container granularity (number of containers), different processor and memory affinity, two different platform architectures (UMA and NUMA), and two application subscription modes (exactly- and over-subscription). The last case conducts a systematical study on the performance of multi-container deployments for HPC workload on a cluster with different network fabrics and protocols, focusing especially on Infiniband networks. We analyze the impact of container granularity and its potential to exploit processor and memory affinity to improve applications' performance (**O1-C1/C2** and **O2-C1/C2**).

- **Chapter 4** presents a systematic study on the performance of multi-container deployment schemes for online ML inference services. We share the findings and lessons learned from conducting representative client loads on an image classification model across numerous deployment configurations, including the impact of container granularity and its potential to exploit processor and memory affinity (**O1-C3** and **O2-C3**).

- **Chapter 5** investigates an architecture for autonomic ML workflows with abilities for multi-layered control, based on an agent-based approach that enables autonomic management and supervision of ML workflows at the application layer and the infrastructure layer (by collaborating with the orchestrator). We design a Scanflow ML framework to support such multi-agent approach by using triggers, primitives, and strategies. In this chapter, we also implement a practical platform, so-called Scanflow-K8s, that enables autonomic ML workflows on Kubernetes clusters based on the Scanflow agents (**O1-C3**, **O3-C1/C2**, and **O4-C1**).

- **Chapter 6** conducts fine-grained scheduling policies for containerized HPC workloads in Kubernetes clusters, focusing especially on partitioning each job into a suitable multi-container deployment according to the application profile. Based on the platform presented in Chapter 5, we implement our scheduling schemes on different layers of management (application and infrastructure), so that each component has its own focus and algorithms but still collaborates with others (**O3-C3** and **O4-C2**).

- **Chapter 7** concludes the thesis and discusses some future work.

- **Appendix A** lists the publications and contributions from the author during this Ph.D., both related and non-related to the thesis.

- **Appendix B** presents grants obtained and activities attended during the Ph.D.

Chapter 2

State of the Art

In this chapter, the introduction of the current state-of-the-art of topics under study in the thesis is provided. The following topics are presented: In Section 2.1, the current HPC, BD, and ML communities are elaborated. For each community, we introduce application characteristics, corresponding software stacks, specific hardware and benchmarks. Section 2.2 explains the virtualization/containerization technology, multi-host container networking, related work regarding multi-container deployment schemes and corresponding container affinity settings. Some discussions about the current container management system and its scheduling mechanisms are described in Section 2.3.

2.1 HPC, BD, and ML Communities

The HPC, BD, and ML communities are different or even separate in both infrastructure and platform layers. The different infrastructures and software stacks for HPC, BD, and ML communities are shown in Figure 2.1. Thus, in the following sections, we introduce each community with its application characteristics, corresponding software stacks, specific hardware and benchmarks.

2.1.1 High Performance Computing

Introduction

With the development of information technology (IT), scientists began to simulate complex phenomena and systems through computation a few decades ago. This computational branch is distinguished from the empirical and theoretical branches as it can solve complex equations for which an analytical solution cannot be achieved. Simulation, the method of doing the computation, is called the third paradigm of scientific development [58]. High Performance Computing (HPC) is required for this simulation for large-scale scientific engineering problems because of its computational speed, accuracy, and supported programming models and libraries. In the third generation of scientific development, HPC is widely used for dealing with complex scientific problems in the areas of Climate prediction [130][179], Bioinformatics, Atmospheric composition, Genomics and Geophysics [1], etc. The supercomputers optimized for HPC are massively-parallel machines usually consisting of thousands of computing nodes with the latest multi-core processors, non-trivial NUMA architectures, computational accelerators (*e.g.,* GPUs), high-speed network interconnects (*e.g.,* Infiniband), and parallel file systems (*e.g.,* GPFS [65], Lustre) [9][150]. These high-performance infrastructures empowering supercomputers directly affect the effectiveness of problem resolutions.

[1]https://www.bsc.es/research-and-development/research-areas

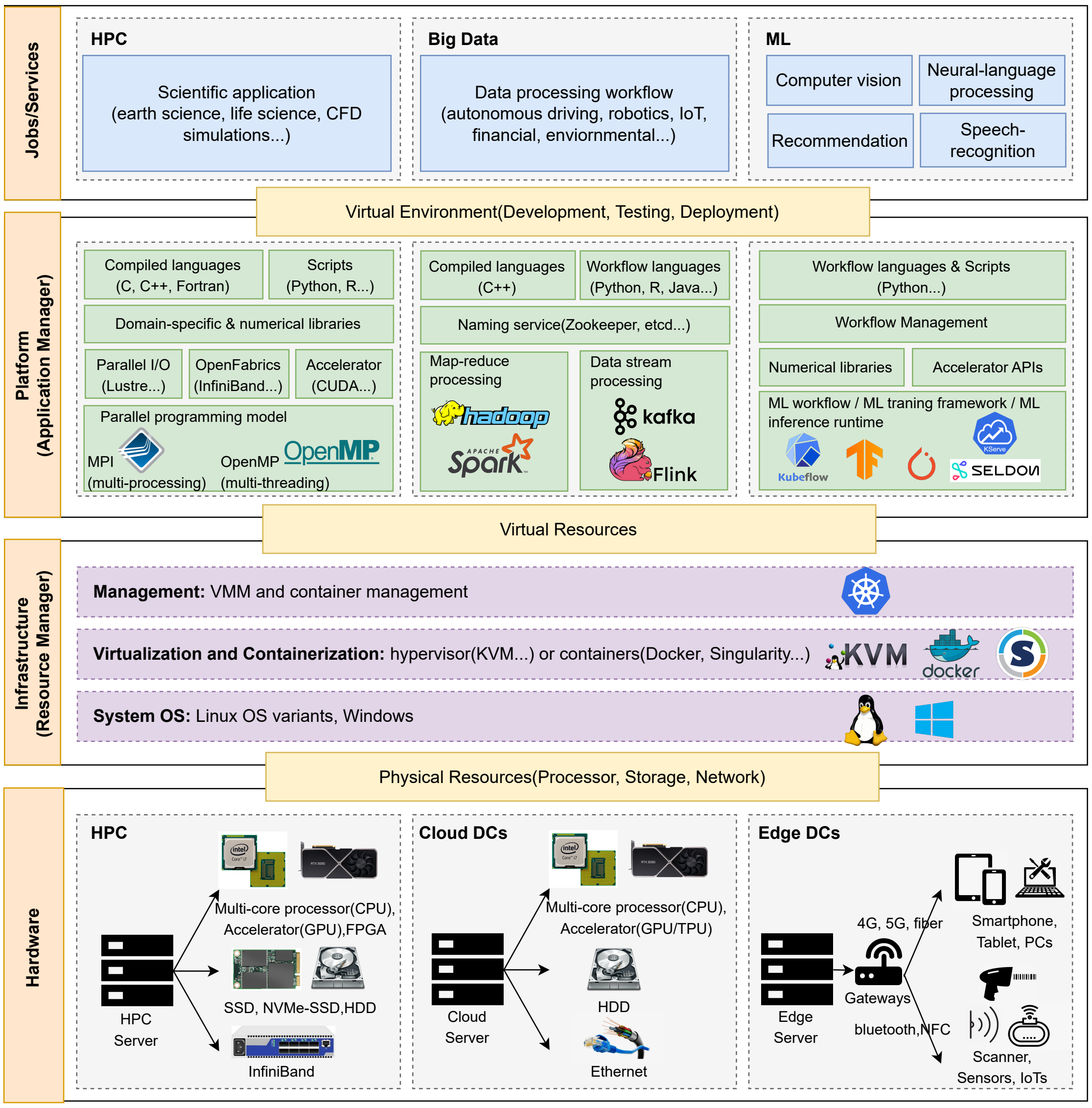

Jobs/Services
HPC
Scientific application (earth science, life science, CFD simulations...)
Big Data
Data processing workflow (autonomous driving, robotics, IoT, financial, enviornmental...)
ML
Computer vision
Neural-language processing
Recommendation
Speech-recognition
Virtual Environment(Development, Testing, Deployment)
Platform (Application Manager)
Compiled languages (C, C++, Fortran)
Scripts (Python, R...)
Domain-specific & numerical libraries
Parallel I/O (Lustre...)
OpenFabrics (InfiniBand...)
Accelerator (CUDA...)
Parallel programming model
MPI (multi-processing)
OpenMP (multi-threading)
Compiled languages (C++)
Workflow languages (Python, R, Java...)
Naming service(Zookeeper, etcd...)
Map-reduce processing
hadoop
Spark
Data stream processing
kafka
Flink
Workflow languages & Scripts (Python...)
Workflow Management
Numerical libraries
Accelerator APIs
ML workflow / ML traning framework / ML inference runtime
Kubeflow
KServe
SELDON
Virtual Resources
Infrastructure (Resource Manager)
Management: VMM and container management
Virtualization and Containerization: hypervisor(KVM...) or containers(Docker, Singularity...)
KVM
docker
System OS: Linux OS variants, Windows
Physical Resources(Processor, Storage, Network)
Hardware
HPC
HPC Server
Multi-core processor(CPU), Accelerator(GPU),FPGA
SSD, NVMe-SSD,HDD
InfiniBand
Cloud DCs
Cloud Server
Multi-core processor(CPU), Accelerator(GPU/TPU)
HDD
Ethernet
Edge DCs
Edge Server
4G, 5G, fiber
Gateways
bluetooth,NFC
Smartphone, Tablet, PCs
Scanner, Sensors, IoTs

Figure 2.1: Convergence of HPC, BD, and ML applications on containerized infrastructures.

Message Passing Interface (MPI) Programming Model

MPI (Message Passing Interface) is a programming model that is widely used for parallel computing. It provides a standard set of routines or functions for message passing between multiple processes, which can run on different processors or computers in a network [39]. Processes communicate with each other by sending and receiving messages through a communication network. MPI provides a set of functions (*e.g.,* MPI_Send, MPI_Recv, etc.) for sending and receiving messages, as well as for synchronization and collective operations (*e.g.,* MPI_Barrier, MPI_Allreduce, etc.).

MPI is commonly used in scientific computing and HPC applications where large amounts of data need to be processed in parallel. MPI can be used with a wide range of programming languages, including C, C++, Fortran, and Python, and is supported by many hardware (*e.g.,* clusters, multi-core CPUs, and GPUs) and software vendors. MPI has some reference implementations, such as MPICH[2] or OpenMPI[3].

MPI provides a high degree of control over the parallelism of an application, which can be important for achieving good performance in parallel computing. OpenMPI, for instance, allows to 1) decide the `hostfiles` to enable the parallelism of an application among multiple hosts; 2) configure the appropriate `rankfiles` to carry out the ranking, mapping, and binding between processes to cores; 3) select between two subscription modes: `exactly-subscribed` mode where OpenMPI can run its message passing engine always in aggressive mode (never giving up the processors to other processes) and `over-subscribed` mode where the OpenMPI engine must run in degraded mode and frequently yield the processor to its peers when idle, thereby allowing all processes to make progress [136]. The awareness of the aggressive or degraded mode of the OpenMPI engine is usually automatic, although the user can use the MCA parameter `mpi_yield_when_idle` to control whether an MPI process runs in aggressive or degraded performance mode [135].

InfiniBand

InfiniBand (IB) [121] is a kind of fabric HPC infrastructure of supercomputers for modern high-speed network interconnects. Six of the top ten HPC supercomputers in the world are accelerated by InfiniBand, and more than 59% Top500 [4] HPC platforms are connected by Mellanox InfiniBand and the Ethernet Solutions [123][120][180]. 200G HDR InfiniBand published by Mellanox on 2018 accelerates 31% of new InfiniBand systems on November's Top500, including the fastest built supercomputer [119][122].

InfiniBand [163] interconnect can provide high throughput and low latency communication across systems for distributed and parallel applications. IB comprises two channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCA provides hardware visibility at the user level for communication. OpenMPI follows the standard of the software stack from the OpenFabrics Alliance for the Remote Direct Memory Access (RDMA) through InfiniBand, which allows processes to access the memory of a remote node process without the CPU intervention [174].

InfiniBand interconnect can also support other communication protocols (see Figure 2.2). For example, TCP/IP network protocol stack can be adapted for InfiniBand

[2] https://www.mpich.org/
[3] https://www.open-mpi.org/
[4] https://www.top500.org/

through TCP/IP over IB (IPoIB) [50]. IPoIB is a Linux kernel module that enables InfiniBand hardware devices to encapsulate IP packets into IB datagrams or connected transport services. When IPoIB is applied, an InfiniBand device is assigned an IP address and accessed just like any regular TCP/IP hardware device [174]. The IPoIB driver supports two modes of operation: datagram and connected. In datagram mode, the IB unreliable datagram transport is used. In connected model, the IB reliable connected transport is used. Although IPoIB cannot achieve better performance than the native RDMA, it still has better point-to-point performance than the original TCP/IP over Ethernet [109][107].

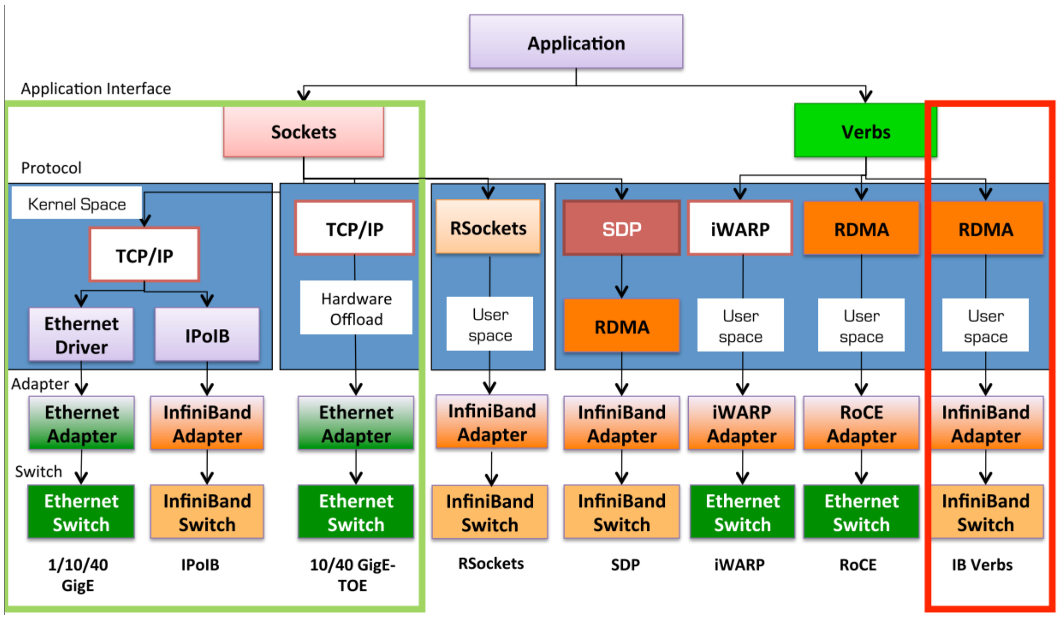


Figure 2.2: Common protocols using Open Fabrics [107][110].

HPC Benchmarks

Some well-known HPC benchmarks are introduced to be used for the thesis evaluations. They are designed to mimic some behaviors present in the real-world HPC applications.

HPC Challenge Benchmark: The HPC Challenge (HPCC) benchmark suite[5] is widely used to evaluate the performance of HPC systems. Its design goal is to enable complete understandings of the performance characteristics of platforms [112]. It consists of several benchmarks that show the performance impact of real-world HPC applications. For example, the capability of processor floating point computation (*e.g.,* DGEMM, FFT), memory bandwidth (*e.g.,* STREAM, FFT) and latency (*e.g.,* RandomAccess), and communication bandwidth (*e.g.,* b_eff, PTRANS, FFT) and latency (*e.g.,* b_eff, RandomAccess, FFT) [197]. They provide common and standard units to evaluate the results of HPCC. The benchmarks are described as follows:

- **EP-DGEMM (DGEMM)** [112]: Real-valued dense matrix multiplication in double precision. Measures the floating point rate of execution in GFLOP/s.

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

- **G-FFT (FFT)** [1]: Global discrete Fast Fourier Transform of a vector. Measures the floating point rate of execution in GFLOP/s.
- **G-PTRANS (PTRANS)** [1]: Global Parallel matrix transpose. Exercises the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communication capacity (in GB/s) of the network.
- **EP-STREAM (STREAM)** [62]: Measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernels.
- **G-RamdomAccess (RA)** [1]: Random memory access. Measures the rate of integer random updates of memory (in GUP/s, *i.e.,* GigaUpdates per second).
- **b_eff** [62]: Measures the latency (in microseconds) and bandwidth (in GB/s) of various communication patterns, including ping-pong and ring.

OSU Benchmark: OSU Benchmark[6] is a suite of benchmarks that measure the MPI-level operation performance. We choose this benchmark for understanding MPI communication performance with different message sizes. For instance, the OSU MPI_Alltoallv Latency Test from the OSU benchmark suite can be used to evaluate the global latency of MPI ranks sending and receiving data and the OSU Bidirectional Bandwidth Test is able to measure the maximum aggregate bandwidth between two adjacent nodes that send out a fixed number of back-to-back messages between them.

MiniFE Application: MiniFE[7] is a proxy application for unstructured implicit finite element codes. It is similar to HPCCG and pHPCCG but provides a much more complete vertical covering of the steps in this class of applications. The physical domain is a 3D box with configurable dimensions and a structured discretization (which is treated as unstructured). The domain is decomposed using a recursive coordinate bisection (RCB) approach and the elements are simple hexahedra. The problem is linear and the resulting matrix is symmetric, so a standard conjugate gradient algorithm is used with a general sparse matrix data format and no preconditioning. The use of this mini-application (small self-contained proxies for real application) is an excellent approach for rapidly exploring the parameter spaces, also it enriches the interaction between application, library and computer system developers by providing explicit functioning software and concrete performance results that lead to detailed, focused discussions of design trade-offs, algorithm choices and runtime performance issues.

2.1.2 Big Data

Introduction

Data is increasing massively over the years [194]. Big Data, defined as high-volume, high-velocity, and/or high-variety information assets has been widely utilized in commerce and finance [45]. Therefore, IT with big data has entered the fourth generation of science development (*i.e.,* Data-Intensive Science) [58]. Those extremely big data from many different sources are facing the challenges of data storage, analysis, and collaboration.

[6] https://mvapich.cse.ohio-state.edu/benchmarks/
[7] https://github.com/Mantevo/miniFE

Some efficient technologies and platforms, such as Hadoop[8] and Spark[9], are supporting this data mining, analyzing, and learning.

Apache Hadoop and Spark

Apache Hadoop is an open-source software library for reliable, scalable distributed computing and a framework that allows for the distributed processing of large data sets across clusters of computers using MapReduce programming models. It contains several modules, including a job scheduling and cluster resource management framework (*i.e.*, Yet Another Resource Negotiator (YARN)), and a distributed file system (*i.e.*, Hadoop Distributed File System (HDFS)) that provides high-throughput access to application data [194].

MapReduce programming model: MapReduce is a programming model for parallel data processing on a large distributed cloud computing environment. MapReduce programs can be written in several languages, such as Java, Python, and Ruby. Details are shown in Figure 2.3.

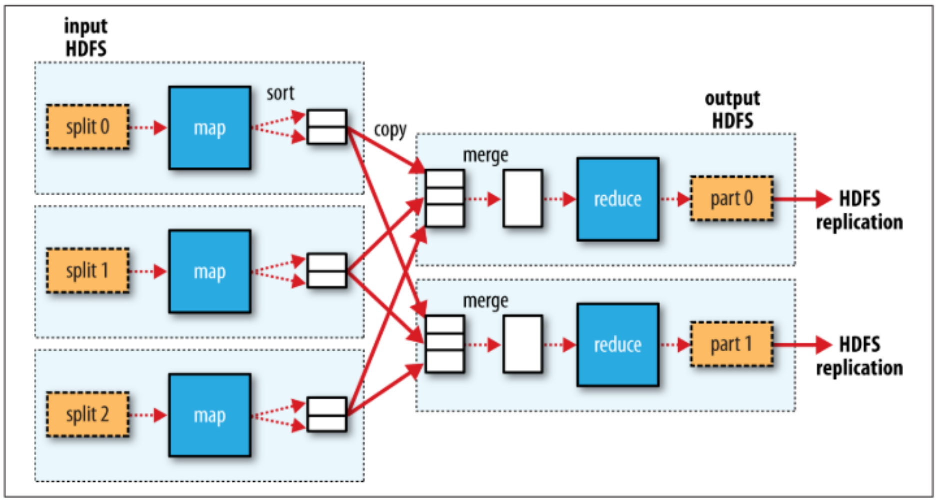


Figure 2.3: MapReduce programming model [194].

MapReduce model works in two stages: the map stage and the reduce stage. Each stage has key-value pairs as input and output [194]. A MapReduce job divides the input data into independent splits which are processed by map tasks in a completely parallel manner. Then, the model sorts the outputs of the maps as the input of the reduce tasks. The reduce tasks fetch the intermediate data from the local or remote file system. After receiving these data from various locations, the reduce tasks merge these data and finally store the results into the file system [5]. Most MapReduce jobs are limited by the network bandwidth available on the cluster because of the data transferred between the map and reduce stages. Also, due to the data being temporarily or permanently stored in the file system, it is better to use the new generation disk drivers with growing transfer speeds.

[8]https://hadoop.apache.org/
[9]https://spark.apache.org/

YARN: Apache Hadoop YARN is a resource management system designed for distributed computing and working on Hadoop clusters. YARN provides a global *Resource manager* as a master to manage the usage of resources (cpu, memory, disk, network) on the whole clusters, *Node manager*, one for each node, who is responsible for starting containers, monitoring, and reporting the resource usage to the *Resource manager*, and an *Application master*, one for each MapReduce job, who is taking charge of negotiating resources from the *Resource manager* and working with the *Node managers* to execute and monitor tasks. YARN scheduler allocates the resources to applications depending on its configured policy, for example, the FIFO, Capacity, and Fair Schedulers.

HDFS: HDFS is a distributed file system designed to run on commodity hardware. It is originally used for the Hadoop framework because of its advantages different from other distributed file systems, such as fault tolerance and throughput. HDFS provides high throughput access to application data and is suitable for applications that have large data sets [4].

Spark: Apache Spark is a unified analytic engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python, and R and high-level tools for a wide range of applications, including MLlib for ML [56], Spark SQL for SQL and structured data processing, Spark streaming for stream processing, and GraphX for graph computation. Nowadays, it also can be deployed on Hadoop YARN [172], Apache Mesos[10] [171] and Kubernetes[11] [57][170][169].

BD Benchmarks

Terasort Benchmark is a commonly used BD benchmark to measure MapReduce performance. In the previous work from our group [160], Terasort was used in the evaluations.

Terasort Benchmark: TeraSort is a popular benchmark that measures the amount of time to sort one terabyte of randomly distributed data on a given computer system. It is commonly used to measure MapReduce performance of an Apache Hadoop cluster. TeraSort combines testing the HDFS and MapReduce layers of a Hadoop cluster and consists of three MapReduce programs.

- **TeraGen** is a MapReduce program that generates large data sets to be sorted.
- **TeraSort** reads the input data and uses MapReduce to sort the data.
- **TeraValidate** validates the sorted output to ensure that the keys are sorted within each file. If anything is wrong with the sorted output, the output of this reducer reports the problem.

2.1.3 Machine Learning

Introduction

Data science and ML are becoming core capabilities for solving complex real-world problems, transforming industries, and delivering value in all domains. ML has become common with good results in different tasks such as image classification, machine translation, recommendation systems, and speech recognition [55][81][111][26]. However, ML is not only about ML algorithms, but surrounds with various and complex elements (as shown in Figure 2.4) for efficient ML model development, training, inference, and maintenance.

[10] http://mesos.apache.org/
[11] https://kubernetes.io/

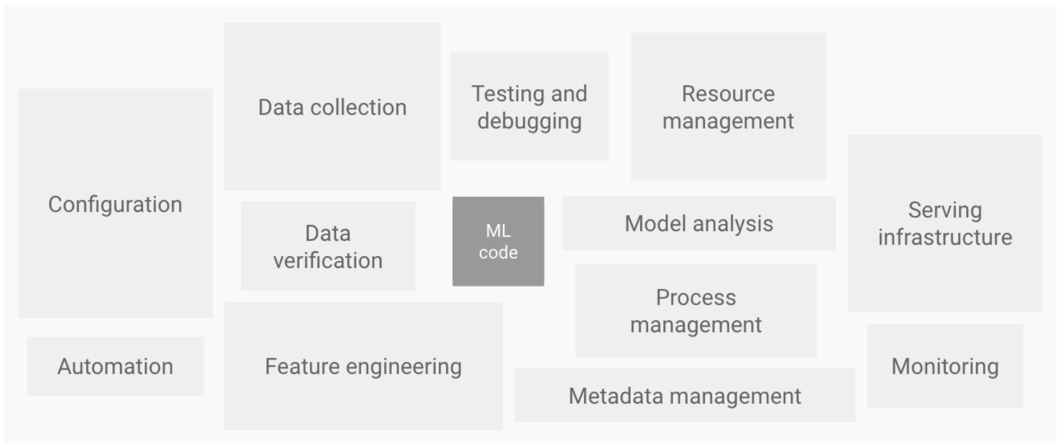


Figure 2.4: Elements for machine learning [49].

ML Workflows

While working on a ML project, workflows comprising some reproducible steps run as a pipeline are widely used to build or deploy a model efficiently because of the flexibility, portability, and fast delivery they provide to the ML life-cycle [48]. The ML life-cycle typically contains two phases, namely *ML training* and *ML inference*.

- **ML Training:** In the ML training phase, ML algorithms are used to build and obtain ML models from a training dataset. A training workflow can be vary depending on specific use cases, but typically involve these steps: *data preparation*, *data validation*, *data preprocessing*, *model development*, *model validation* and *model testing*. From a runtime perspective, the ML training workflow is run as offline jobs and may take days to complete. In particular, the executors of batch workflows run once for each time the workflow is executed. The Data Science team uses the batch ML workflows to build and gain ML models at the ML training phase.

- **ML Inference:** Once the model has been evaluated and its performance is satisfactory, it can be deployed to a production environment. In the ML inference phase, given ML models are used to make predictions from new data. ML inference workflows normally have these steps: *data validation*, *data preprocessing*, and *model serving*. ML inference workflows can be either conducted as a batch process, where predictions can be generated asynchronously from a batch of samples with no specific time limit to receive the prediction results, or through an online ML inference service, which receives dynamic queries from end-users and serves the predictions in real-time (subject to a latency bound) [97][98][203][92].

  From a runtime perspective, the batch ML inference workflow is run as offline jobs and may take days to complete, whereas the online ML inference service is realized as a long-run service that is able to deal with dynamic queries issued by end-users. In particular, the online workflow is deployed as a long-run microservice that is able to deal with clients' invocations by APIs. The Data Engineer team conducts the batch ML workflows for batch predictions or deploys online ML workflows in production to make real-time predictions at the ML inference phase.

ML Frameworks

Many tools and frameworks are available to help streamline ML workflows and make the operations easier to develop, train, and deploy ML models.

There are many popular ML frameworks and libraries available to build ML models, such as TensorFlow [175], Keras, PyTorch [143], and Kubeflow Pipelines[12] that make it easier to develop and train ML models. Lately, many data researchers and companies have been interested in automating the ML tasks within a training workflow (*e.g.,* AutoML) in order to construct ML models efficiently, such as model selection, hyperparameter tuning, and feature engineering [199][155][19][38][77]. It aims to simplify the process of building and deploying machine learning models by automating some or all of the steps involved in the ML workflow. However, these powerful AutoML modules and frameworks are turned off after training a model, thus cannot help the model after being deployed to meet dynamic changes.

Once the model is ready, the next step is to deploy it into a production environment. ML models are trained through diverse ML frameworks, which causes the runtime of the ML inference services to be different. Early works on ML inference systems such as Clipper [28] and Rafiki [191] deployed models in containers using custom runtime and implemented an abstract layer between clients and models to achieve model selection and request batching. Currently, there are several open-sourced runtime for online ML inference services in production, such as Tensorflow serving [176], TorchServe [144], Kserve [82], or Seldon [161], as well as some optimized libraries for ML, such as Intel Math Kernel Library (MKL) [71]. These runtimes may vary, but they contain similar functionalities (*e.g.,* model version management, model warmup) and configuration settings (*e.g.,* parallel threading model, batching, and caching) [176]. Experienced Data Engineers could tune the parameter settings of the model serving runtime to improve the performance. For instance, Hasabnis et al. studied how to auto-tune the threading model for Tensorflow serving and MKL CPU backends [54].

In addition to developing, training, and deploying the ML model, once the model is deployed, it is also important to continuously monitor and maintain the model to ensure that it continues to perform well. This can involve tasks such as retraining the model on new data, updating the model, and monitoring the model's performance and accuracy over time. Those related works regarding autonomous management and supervision for ML applications will be introduced later in Section 2.3.3.

ML Usecases and Benchmarks

ML usecases are used for prototyping a ML model from training to inference in production. Handwritten digits classification with MNIST dataset is a widely used example for fast prototyping. Other ML models can be adopted in the same manner as the MNIST usecase. MLPerf Inference is a benchmark suite for measuring how fast systems can run models in a variety of deployment scenarios. We use it in the performance analyses for online ML inference. Also, we use its client to generate different realistic loads to create different self-adaptation scenarios.

MNIST Usecase: MNIST[13] is a popular dataset in machine learning used for image classification tasks. It consists of 70,000 grayscale images of handwritten digits (0-9)

[12] https://www.kubeflow.org/
[13] http://yann.lecun.com/exdb/mnist

with 28x28 pixel resolution. This dataset is often used as a benchmark in the development and testing of machine learning algorithms for image recognition and classification tasks. It has been used extensively in academic research, industry, and education to train and evaluate machine learning models. One of the most common use cases of the MNIST dataset is to train machine learning models to recognize handwritten digits. The goal of this task is to correctly classify the digits in the images. In addition to image classification, MNIST is also used for testing and benchmarking other machine learning algorithms, such as anomaly detection and generative models.

MNIST-C[14] is a dataset that was created as a variant of the original MNIST dataset. The "C" in MNIST-C stands for "Corrupted", as the images in this dataset have been artificially corrupted to simulate common real-world image corruption and distortion. The goal of using MNIST-C is to evaluate the robustness of machine learning models to various types of image corruption and distortion. Machine learning models that can perform well on MNIST-C are more likely to perform well in real-world applications where the images may be distorted or corrupted.

MLPerf Inference Benchmark: MLPerf Inference Benchmark[15] is a benchmark suite specifically designed to measure the performance of ML models during inference. It includes standard models, datasets, and evaluation metrics of different client scenarios, which enables fair and comparable measurements. Additional benchmark details can be found in [149].

- **MLPerf Model and Dataset**: MLPerf provides computer vision applications with its associated reference model (*i.e.,* a classifier network takes an image and selects the class that best describes it). In particular, for image classification, it provides a well-known vision model: the computationally-intensive Resnet50 [55] as a benchmark. This model accepts base64-encoded JPEG images as input and decodes them within the inference stage. We use the ImageNet 2012 dataset, crop the images to 224x224 in preprocessing, and send the strings of base64-encoded images through the physical network using REST APIs.
- **MLPerf Client Scenarios**: MLPerf LoadGen provides four realistic end-user scenarios, namely Single-Stream (SS), Multi-Stream (MS), Server (S), and Offline (O), which represent many critical inference applications. Figure 2.5 shows how LoadGen generates queries for each scenario.
  - SS: The *Single-Stream* scenario represents the client sending inference-query streams one by one (*i.e.,* the client waits for the completion of one query before issuing another) with a query sample size of 1. The objective is to assess the responsiveness of the SUT by means of the 90th percentile latency.
  - MS: The *Multi-Stream* scenario represents the client sending inference-query streams with a fixed time interval. It assesses the maximum query sample size of each inference-query stream subject to a latency bound. No more than 1% of queries may exceed the latency bound.
  - S: The *Server* scenario represents an application where the one sample-sized inference-query streams are arriving randomly at the SUT with a Poisson distribution. The SUT responds to each query within a benchmark-specific

[14] https://github.com/google-research/mnist-c
[15] https://github.com/mlcommons/inference

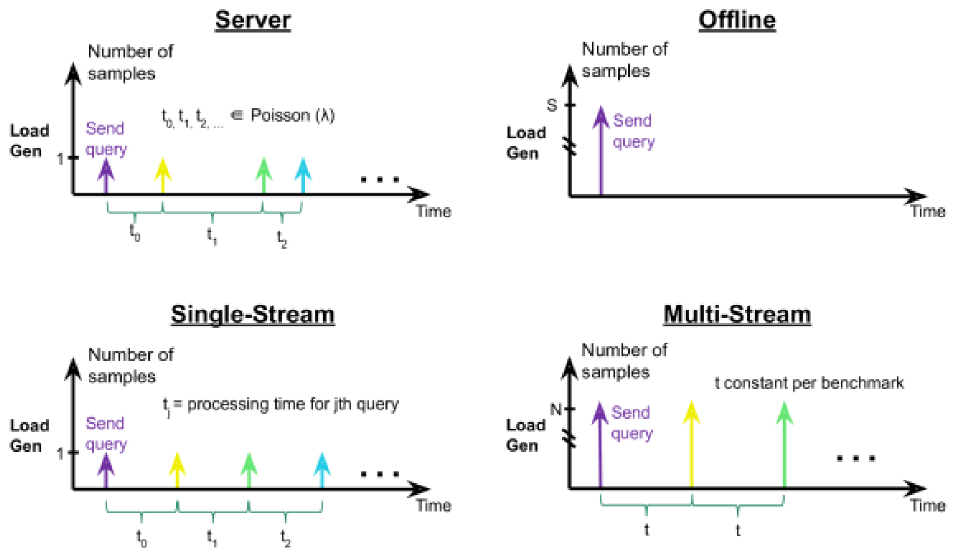


Figure 2.5: The timing and number of queries from LoadGen [149].

latency bound. No more than 1% of queries may exceed the latency bound. The performance metric is the Poisson parameter that indicates the queries-per-second (QPS) achievable while meeting the latency QoS requirement.

– O: The *Offline* scenario represents batch-processing applications where all the data are sent to the SUT as soon as possible and latency is unconstrained. The performance metric is the throughput measured in samples per second.

2.2 Virtualization/Containerization Technology

Virtualization/Containerization technologies are typically used in data centers and cloud environments, and decouple applications from their relied infrastructures [165]. Section 2.2.1 introduces the traditional and mature hardware virtualization technology and its use to support HPC, BD, and ML applications. An emerging containerization technology and its implementations are described in Section 2.2.2. Section 2.2.3 introduces multi-host container networking, including underlay and overlay network approaches. Related work regarding multi-container deployment schemes and their corresponding affinity settings for different applications and hardware are presented in Section 2.2.4.

2.2.1 Hardware Virtualization

Virtualization technology has been developing rapidly over the few decades because vendors, including Amazon, Google, Microsoft, etc., have been growing interested in providing their computing infrastructures as cloud services. As a result, virtualization has become one of the basic technology of cloud computing [204]. Hardware Virtualization solutions, firstly used on production, such as VMware[16], VirtualBox[17], Xen[18]

[16]https://www.vmware.com/products/esxi-and-esx.html
[17]https://www.virtualbox.org
[18]https://xenproject.org/

and Kernel-based Virtual Machine (KVM)[19] have been improving by enterprises and the open-source community. These hardware virtualization solutions support the customization and portability of the application's computing environment and the execution of the application in a secure and isolated way from other applications sharing the same platform.

Hardware virtualization, namely abstracts the hardware and system resources from the host operating system for instantiating virtual machines that act like real computers with their own operating system and kernel. Those virtual machines are typically referred as guests, and concurrently run on a host supervised by a hypervisor. Some researches discover the possibility of virtualizing the HPC platforms [184][126]. Concluding from Younge et al, [201], KVM is the best overall choice for HPC cloud environments. Thus we expect other hardware virtualization technologies using the same hardware acceleration features are providing a worse or similar performance to KVM.

Linux KVM is a kernel module, included in the mainline, that lets the processor on the host machine enter into a guest state. The guest system has its own set of ring states, but privileged ring0 instructions fall back to the hypervisor code. Furthermore, the KVM module also handles low-level parts of the emulation, like the MMU registers, PCI emulated devices. QEMU[20] is a standalone software that emulates machines or as in an official definition, a generic open-source machine emulator and virtualizer. In this case, emulation means that binary code written for a given processor is recompiled to run for another one. QEMU and KVM work together to fully virtualize a virtual machine. Libvirt[21] interacts with the underlayer of KVM and QEMU and provides a high-level user interface for the user's operation in order to easily manage VMs.

Several studies have explored the evaluation of the hardware virtualization performance in HPC environments [201][165][37]. In general, hardware virtualization technology has the extra operating system and the hypervisor enhancing large latencies and overheads, thereby deploying applications on it has fallen short to reach the performance of bare-metal executions. VMware [185] designs hardware virtualization work together with HPC platforms to deliver a secure, elastic, fully managed, self-service, virtual HPC environment running HPC workloads. It admits that performance degradation of MPI workloads due to latency requirements combined with intensive communication among processes. However, they also propose the VM can achieve performance near bare metal by leveraging computer accelerators. Also, the usage of hardware virtualization for BD and ML applications has been studied in several works [73][29][147]. Some point out promising advantages in some scenarios, for instance, to exploit NUMA locality by running several virtual machines in a physical host and sizing them in order to fit within a single NUMA node [29].

2.2.2 Containerization

The emergence of operating-system-level virtualization (*i.e.,* containerization) alleviates performance problems while maintaining most of the advantages of virtualization [165][37]. Each container environment and isolation are managed by the host kernel, which allocates the needed resources (CPUs, memory, network I/O) to each container.

[19]https://www.linux-kvm.org/page/Main_Page
[20]https://www.qemu.org
[21]https://libvirt.org

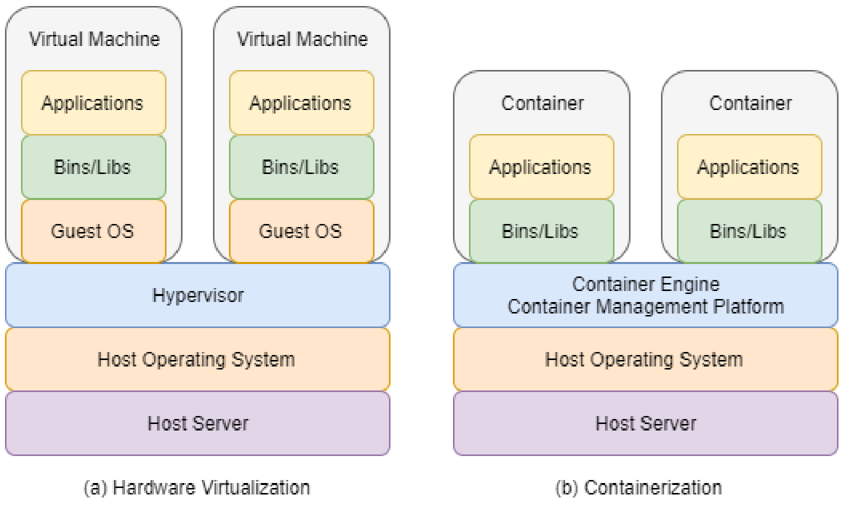


Figure 2.6: Difference between Hardware Virtualization and Containerization.

The differences between hardware virtualization and containerization are shown in Figure 2.6. The key point of containerization is that containers do not run their own kernel, but share the underlying host kernel for OS services and run a different software stack. Also, containers directly communicate with the OS by the system calls whereas the VMs run in a non-privileged mode and the instructions inside need to be translated into the host instructions by the hypervisor.

Recently, containerization such as Docker[22], Linux Containers (LXC)[23], OpenVZ[24], Podman[25], Shifter[26] and Singularity[27] become alternative solutions to hardware virtualization because of its performance and agility [208]. Therefore, enterprises and communities have been getting more interest in using them for the fast, portable and flexible deployment of HPC, BD, and ML applications [70][200][17][209][208].

Docker

Docker, the most popular containerization technology, builds upon resource isolation and limitation features of the Linux kernel, such as `namespaces` and `cgroups`, respectively. Also, it adds a union-capable file system such as Overlay Filesystem (OverlayFS).

Without the hypervisor needed for virtual machines, Docker contains a lightweight engine to control and manage its containers. Also, Docker allows containers to share the underlying host kernel including the libraries, modules, kernel functions, and a root file system. Regarding runtime isolation, Docker containers are defined into some operational spaces (*e.g.,* Network, Process Identifier (PIDs), User Identifier (UIDs), Inter-Process Communicate (IPC)) which are implemented by means of `namespaces`. Regarding resource limitation, some sets of dedicated resources that are defined by means of `cgroups` can be allocated to the Docker containers.

[22]https://www.docker.com/
[23]https://linuxcontainers.org
[24]https://openvz.org/
[25]https://podman.io/
[26]https://shifter.readthedocs.io/en/latest/#
[27]https://sylabs.io/

Docker has advantages including portable and flexible deployment, registry, scalability of applications and is being supported by different frameworks and can be used for running BD and ML applications [70][200][209][17][208].

Singularity

Singularity containers are mostly used in HPC environments where they are proven to introduce less overhead than Docker while providing more reliable security guarantees [8]. Regarding security, Singularity creates containers as individual entities that are executed by the host system instead of creating containers as spawned child processes of a root owned daemon. Regarding performance, Singularity enables all the containers to use the underlying HPC environment in a natural way (without namespaces isolation). Because of this feature, the integration between Singularity and MPI can be transparent to the user. Users only need to run `mpirun` command as they run it on bare-metal machines, then the MPI process management daemon (ORTED) will handle the containers execution and the processes launching and communications. These make Singularity a first-class choice for HPC and scientific simulations [91][159].

In late 2018, Singularity 3.0 was released [63]. This version brings a new functionality (so-called instances) to run containers in “daemon” mode, which allows running them as services in the background. Singularity instances can have isolated network resources, and they also support cgroups functionality to restrict the resource usage. MPI applications can run in Singularity instances as if they were running in separate hosts, having its own network identity and using an SSH backend service to communicate. In this sense, Singularity instances somehow mimic Docker, while still keeping the advantages regarding security, thus we also include them as a part of our evaluation.

2.2.3 Multi-host Container Networking

Communication inside a container uses shared memory, because all the processes have the same IPC namespace which shares the IPC resources like message queues, semaphores, and shared memory. Containers within a same host use the host network to communicate. Moreover, containers could communicate across hosts through both *underlay* and *overlay* networking approaches.

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

In overlay network approaches, a logical network between the containers is built using networking tunnels to deliver communication across hosts. Those tunnels add an additional level of encapsulation to the underlying network. Because of this, they may introduce some extra overhead when compared with an underlay approach, due to the encapsulation overhead of the frame size and the processing overhead on the server. Nevertheless, overlay network approaches are very flexible as they decouple the virtual network topology from the physical network, which supports for instance the mobility of components independently of the physical network. In addition, they essentially support an unlimited number of components, as they do not suffer from restrictions to the number of addresses imposed by the physical network [160].

2.2.4 Multi-container Deployment Schemes and Affinity Settings

In addition to using different container technologies and container networks, container granularity and container affinity settings show other aspects that can be utilized to improve application performance. In the previous work in our group [160], we enabled the interconnection across hosts through TCP/IP protocol between Docker and Singularity instances running a BD application, and also analyzed the performance of enabling affinity for BD workers. In this section, we focus on the related work shown in the field of using multi-container deployment schemes and affinity for HPC or ML workloads.

Multi-container Deployment Schemes for HPC Workloads

In order to improve the performance of HPC applications on virtualized multi-socket architectures, several works have proposed partitioning the HPC applications into several virtual machines to prevent them spanning multiple NUMA domains [66][67]. The same idea of sizing virtual machines conforming with NUMA boundaries for throughput workloads (*i.e.,* MPI jobs without communication) has been suggested by VMware in their reference architecture for virtualizing High Performance Computing [185].

Consequently, application partitioning comes together with the need to schedule the resulting virtual machines in the NUMA platform so that each of them optimizes its memory access locality [148][23] or the access to local I/O devices [14]. Cheng et al. [23] presented a user-level scheduler that periodically adjusts the placement of virtual machines aiming for local node execution, that is, the VCPUs of a virtual machine are running on one NUMA node and its memory is also located on the same NUMA node. Rao et al. [148] proposed a load-balancing algorithm to determine the optimal VCPU-to-core assignment by dynamically migrating VCPUs to minimize the penalty to access the uncore memory subsystem.

However, there is few empirical research yet evidencing whether the experience of virtualization can be applied to containerization with the same benefit for HPC applications. Yang et al. [198] proposed a management service for Docker containers based on OpenStack, which features a NUMA-aware mechanism that limits the accessible CPU and memory of containers to the same NUMA node. However, this work does not consider multi-container deployments from a single tenant but a simple scenario with two containers from two different tenants.

Several studies have compared the overhead of using virtualization and containerization technologies for HPC applications [16][165][178]. They claimed that containerization has less overhead than virtualization in most cases. As a result, many works have focused on the performance analysis of containerized deployments for HPC applications. Xavier et al. [196] firstly did in 2013 a full performance comparison of container-based technologies relevant at that time, mainly Linux-VServer, OpenVZ, and LXC, for HPC workloads. However, containerization technologies have evolved considerably since then, and new ones must be also evaluated for deploying HPC applications. In particular, Docker has become the most popular containerization software, and Singularity is also widely used in the community to support HPC workloads [8][154][192][202]. These works have focused on evaluating the performance of an HPC application from a single-tenant on a single container allocated on a single node with different containerized technologies. Other works considered multi-tenant HPC workloads. Maliszewski et al. [114] investigated the performance of scientific workloads with single or multi-tenant instances in a single node, where each tenant held its independent application among other tenants. Jha et al. have studied HPC microservices in different container environments [74, 75]. Their work includes flexible deployments for HPC applications on a single node, from running a single or multiple applications in a single container, to running multiple containers each holding a single application. Whereas these studies above considered one or multiple tenants, co-located independent applications on a single node, or allocated one or more applications into a container, none of them considered a deployment scheme partitioning one application into several containers.

Other works have evaluated HPC workloads in distributed containerized platforms. Saha et al. [158] evaluated the performance of an HPC application running with several processes distributed across multiple containers using Docker Swarm, and studied different network methods, number of hosts, and ranks per container. Some of their results showed that one rank per container had degradation, but they did not explain in-depth why this occurred. In another work [157], the same authors presented a framework combining Apache Mesos and Docker Swarm which can orchestrate distributed containers with MPI processes across the nodes. They studied the overhead of running a different number of MPI processes and nodes, and presented a co-scheduling policy. These works showed that there is a possibility that distributed containers with partitioned processes from a single HPC application can be allocated on the same host. However, they mainly focused on the overhead of the orchestrator and the number of nodes, and did not study the specific interference among these containers while being allocated on the same node.

Chung et al. [25] considered the container granularity. Their work studies the scalability of running an HPC application on one or more containers. However, they only measured Docker performance for computation and data access intensive HPC applications, and did not distinguish the different subscription modes of the application or compare different containerization technologies.

None of these works study the joint impact of container granularity and processor and memory affinity settings for multi-container deployments. Furthermore, none of them feature either an in-depth performance evaluation of Singularity, including its instance-based variant, and also a scenario adding CPU cgroups to its original implementation. Thus, we contribute a performance comparison and analysis of distinct multi-container deployment schemes for HPC workloads in Sections 3.3 and Section 3.4 of Chapter 3.

Multi-container Deployment Schemes for HPC Workloads with InfiniBand

Recent works have evaluated Docker and Singularity as candidate containerization technologies to run HPC applications [8][202]. These works mainly focused on a single container wrapped HPC application allocated on a single host, but without considering different container granularity and different container interconnects. Rudyy et al. [154] discussed the execution of a given containerized HPC application on HPC clusters, and mainly studied different container technologies and different HPC architectures, but did not consider different container granularity.

Zhang et al. [205][204] studied the performance characterization of KVM and Docker for running HPC applications on SR-IOV enabled InfiniBand clusters, and in a further work [207], they stated that Singularity-based container technology is ready for running MPI applications on HPC clouds. Also, in their work [206], they studied the locality and NUMA-aware MPI runtime for nested virtualization (a combination of virtual machines and containers). These works evaluated different aspects of using containerization for HPC applications, but none of them considered deployment schemes with different container granularity.

Chung et al. [25] evaluated Docker containers for deploying MPI applications. They proposed deployment scenarios with different container granularity. However, this work only tested computing-intensive and data-intensive applications and did not consider InfiniBand networks. Their further work [24] considered Docker on InfiniBand and highlighted the benefits of using InfiniBand with Docker. This work showed the results of several benchmarks, but did not consider affinity or different network fabrics and protocols.

Saha et al. [158] evaluated the performance of running HPC applications using Docker Swarm. Whereas they considered a different number of MPI ranks distributed in multiple containers across multiple hosts, their latency experiments only include a fixed message size (*e.g.,* 65536 bytes). Their results showed that deploying one rank per container had worse performance because they ignored the binding policy.

Saha et al. [157] enabled the orchestration of MPI applications with Apache Mesos, and provided a policy-based approach for deploying MPI ranks on containers with different granularity. However, this policy is based on TCP/IP over Ethernet, and does not consider InfiniBand. Beltre et al. [15] evaluated Kubernetes to run MPI applications in clouds. They compared TCP/IP and InfiniBand, but they did not include multi-container deployments.

In the previous work in our group [160], we enabled the interconnection across hosts through TCP/IP protocol between Docker and Singularity instances running a BD application. Moreover, in Section 3.3 and Section 3.4 of Chapter 3, we performed a performance analysis of multi-container deployments with different container granularity for a number of HPC applications, but only with a single host and the TCP/IP protocol.

Existing literature shows approaches and results of deploying a single container per host using Docker or Singularity in the cloud, and most of the work considers using the orchestration thus ignoring the original impact of the network fabric and protocols. Moreover, there still exists a gap in terms of multi-container per host deployments evaluation on an InfiniBand cluster which considers the performance of different container granularity and enhanced affinity settings using different network fabrics and protocols for HPC workloads. Thus, we contribute a performance comparison and analysis of distinct multi-container deployment schemes for HPC workloads on InfiniBand cluster in Section 3.5 of Chapter 3.

Multi-container Deployment Schemes for ML Workloads

Online communities have shared some lessons regarding how different settings can impact the performance of deployments. Park and Paul from Tensorflow tested a Tensorflow Serving deployment of an image classification model across numerous deployment configurations, such as different infrastructures, trade-offs between more or fewer servers (but by using different sizes of the virtual machines), number of threads for deployments, and dynamic batching considerations [140]. Morgan et al. studied the batch size and core count scaling for the BERT-like model, as well as manually tuned multi-stream and affinities [128][129]. These works present different deployment and scaling options, but they do not directly assess multi-container deployments. Moreover, their evaluation does not consider realistic client scenarios.

Multi-container deployments have been studied by some authors, although they have not considered online ML inference services. In particular, Medel et al. [118] conducted a performance analysis over Kubernetes considering the deployment and initialization overhead as well as understanding the performance of different pod settings. Moreover, they provided a rule to decide the number of containers per pod by considering the characteristics of the application. Our work in Chapter 3 demonstrated through standalone executions that some types of containerized HPC applications achieve better performance when exploiting multi-container deployments which partition the processes that belong to each application into multiple containers in each node, and when constraining each of those containers to a single NUMA (Non-Uniform Memory Access) domain or pinning them to specific processors. Those multi-container deployments have been demonstrated to improve the performance of HPC workloads comprising loosely-coupled CPU-intensive processes, which resemble the characteristics of ML inference services. This served as an inspiration to explore as well these schemes for online ML inference services.

General approaches for infrastructure-layer autoscaling of online services on the Cloud have been also proposed [104][190]. Moreover, some works have focused specifically on the deployment and scaling of online ML inference services. MArk (Model Ark) [203], a low-latency, cost-effective inference serving system on the Cloud, used predictive scaling to mask the instance provisioning latency. PRETZEL [92] opened a black box of a model-serving application and enabled model-specific optimization with resource sharing. Nexus [166] performed detailed scheduling of GPUs for DNNs. Its design enabled several optimizations in batching and allowed more efficient resource allocation. Swayam [51] derived a global state estimate from the local state and employed a globally consistent protocol to proactively scale-out service instances for SLA compliance, and passively scale-in unused backends for resource efficiency. However, all these works mainly focus on infrastructure resource scaling to satisfy the SLAs and save costs, not considering container count scaling in a host.

The multi-container deployments and affinity schemes provide an additional dimension of deployment configurations at container-level for online ML inference services, thus these schemes could be seen as complementary to infrastructure-layer scaling approaches and could be used together to optimize the performance of online ML inference services. This results in the work in Chapter 4.

2.3 Container Management and Orchestration

The development of container technology has provided support for packaging and operating HPC, BD, and AI workloads. Deploying different types of workloads using containers and running on multi-tenant cloud platforms is happening. In order to automate the deployment and management of those containers with different features, the management tools are required to be adaptive and improve the flexibility, availability and fault-tolerant of the workloads upon.

2.3.1 Container Orchestration Platform

Currently, companies and communities have been rapidly adapting the container to be the base technology for cloud computing as an alternative to VM. Thus, a containerized deployment and operation management platform is their new effort to be achieved. Companies provide some secure, reliable, and scalable orchestration solutions, such as Google GKE (Google Kubernetes Engine)[28], Amazon ECS (Amazon Elastic Container Service)[29], and Amazon EKS (Amazon Elastic Kubernetes Service)[30]. Some open-source alternatives contributed by the community are Kubernetes[31] or Apache Mesos[32] and also Docker provides its native Docker-swarm[33] for container management. These platforms can support containerized workloads, and each has its own characteristics in terms of container management, deployment, scheduling, scaling, and migration.

Table 2.1: Comparison of some container orchestration platforms.

	Docker-swarm	**Kubernetes**
Cluster deployment	Support	Support
Automatic deployment	Support	Support
Service discovery	ETCD/Consul/ZooKeeper	ETCD
Load balancing	Support	Support
Multi-host Network	Overlay	Flannel,Calico
Persistent storage	PersistentVolume	Volume
Scaling	Support	Support
Scheduling	Filter [32] and strategy [31]	Predicates and priorities [87]
Migration	Support(offline migration)	Support(offline migration)

The specific comparisons between two open-source platforms are shown in Table 2.1. Docker Swarm scheduling framework consists of filter [32] and strategy [31]. A filter component tells the scheduler which nodes to use when creating and running containers. Users need to set the labels of the nodes according to their functions before creating containers. Once running containers, the scheduler can filter a set of suitable nodes through the labels for the next step of strategy selection. Strategy component has three

[28]https://cloud.google.com/kubernetes-engine/
[29]https://aws.amazon.com/ecs/?nc1=h_ls
[30]https://aws.amazon.com/eks/?nc1=h_ls
[31]https://kubernetes.io/
[32]https://mesos.apache.org/
[33]https://docs.docker.com/engine/swarm/

strategies: spread, binpack, and random. The spread and binpack strategies compute rank according to a node's available CPU, its RAM, and the number of containers it has. The random strategy uses no computation. It selects a node at random and is primarily intended for debugging. Native scheduling strategies have some shortcomings, for example the spread cannot make full use of the resources, the binpack is easy to have overloads and the random cannot control by users. Some works have improved the scheduling strategies for balanced allocation from multi-objects in order to achieve efficiency of the clusters and the users' requirements [93][94][96][124].

As Docker Swarm is generally used for managing Docker containers, Kubernetes, based on Google's many years of container management experience, becomes a universal container orchestration platform to manage all kinds of containers that implement a CRI (Container Runtime Interface). It can quickly and predictably deploy applications, scale applications, migrate applications, and seamlessly evolve and upgrade. Kubernetes scheduling also has two stages: predicates and priorities [87][85]. The predicates stage is used for filtering unsuitable nodes by using labels or resource conditions. The priorities stage is to score the possible selections by some priority algorithms and choose the highest scored strategy. In production, users can combine those strategies to achieve their own requirements, also Kubernetes provides configuring multiple schedulers [84] and stopping choosing feasible nodes to tune better performance in large clusters [86]. Extending the Kubernetes scheduler is studied in several works [117], most of them focus on making the full utilization of the resources and achieving the user's requirements.

Instead of the default Kubernetes scheduler, Kubernetes toolkit ecosystem also provides other schedulers to enhance the capability of Kubernetes scheduler. For instance, Volcano [186] is an add-on batch scheduling system for computation-intensive workloads on Kubernetes. It features batch scheduling capabilities (such as gang scheduling to make sure that a job will start to run only when all its tasks are ready to be deployed) that Kubernetes scheduler does not support, and also integrates some HPC/BD/ML domain frameworks in its controller. It also features a customizable scheduler so that the system operator can choose different strategies for job scheduling.

2.3.2 Enabling Containerized Applications in Kubernetes

In chapters 4-6, containerized applications are enabled in Kubernetes clusters. We take advantage of Kubernetes' toolkits and ecosystem. In the application layer, the application specification and control can be extended using a Kubernetes Operator containing Custom Resource Definition (CRD) and a custom controller. For this, we particularly focus on enabling an application's multi-container definition. In the infrastructure layer, Pods of each application are being scheduled by Kubernetes default scheduler and launched by Kubelet node agent. The enhancement is also needed in the scheduler to consider also the application information while scheduling and in the node agent as well to consider the affinity.

Orchestration of Containers in Kubernetes

Containerization is a lightweight virtualization technology that builds upon resource isolation and limitation features of the Linux kernel, such as `namespaces` and `cgroups`, respectively. Currently, containerization is widely used to pack applications because of its portability, isolation, and high availability. Generally, there are two types of containerized applications running in the Cloud.

- Long-lived online microservices: loose-coupled services, each of them being an independent module to be deployed or managed in the long term (*e.g.,* Web services).
- Short-lived batch jobs: batched processing jobs, each of them comprising a batch of tasks that are executed once and then terminated (*e.g.,* MapReduce, MPI, and Spark).

Kubernetes supports both types of applications. From the users' perspective, they submit their specifications of services or jobs to Kubernetes, which is responsible for encapsulating them into containers that are wrapped in Pods to be deployed in the nodes. From the providers' perspective, all the nodes and resources are controlled by Kubernetes. Whenever there is a request, Kubernetes managers have to generate the Pod specification for each type of job or service, and select the node (using a scheduling policy to filter and rank nodes) to run each Pod, so that the Kubernetes node agent (*i.e.,* Kubelet) can launch the Pod in the selected node.

Enabling BD Workloads in Kubernetes

BD workloads can be managed by Kubernetes. For instance, Spark provides a driver to generate executors to run BD applications in Kubernetes. Kubernetes will be the cluster manager, in a fully supported way on par with the Spark Standalone, Mesos, and Apache YARN cluster managers [7].

Currently, the Spark driver will start executors directly within Kubernetes pods, connects to them and executes application code. Also, it makes use of the native Kubernetes scheduler. The challenges will be the lack of the scheduling features such as dynamic resource allocation, external shuffle services, job queues and resource management. The experimental solution could be using other customized schedulers for Spark on Kubernetes such as Volcano [34] and YuniKorn [35].

Enabling HPC Workloads in Kubernetes

HPC workloads are considered as batch jobs in Kubernetes. An HPC workload is specified as a launcher and one or multiple workers. Each launcher or worker is a container that can be executed as a Pod and run in parallel in a Kubernetes cluster. However, the original Kubernetes batch jobs are not designed to support the HPC applications efficiently. The specification for HPC applications is limited, thereby relevant application-related information cannot be considered while scheduling. Also, the Kubernetes default scheduler does not schedule jobs but individual Pods. Thus, some add-ons have been designed by the community to enhance usability when specifying and allocating HPC workloads.

Kubeflow MPI operator [83] provides a better specification for MPI jobs which defines an MPI 'Launcher' and an MPI 'Worker'. In most cases, all the MPI worker processes will be launched in this worker container. Moreover, Kubeflow MPI operator mounts the ssh folder for all Pods belonging to the job through a Kubernetes Secret to establish the communication. But this operator does not enhance the Kubernetes default scheduler,

[34]https://volcano.sh/en/
[35]https://yunikorn.apache.org/

thus the allocation of Pods is not considered at the MPI job level but for each individual Pod. Volcano [186] is an add-on that could support HPC workloads on Kubernetes. It provides ssh/service plugins to deal with the Pods' connection and permissions and with the service discovery. It is also a customizable scheduler, so that the system operator can choose different strategies for job scheduling.

Enabling ML Workflows in Kubernetes

ML workflows contain ML training and ML inference phases. ML training workflows can be generated by using some tools such as Kubeflow[36], Argo[37], etc. Those frameworks could have a container support, where each step can be run inside a container. Several frameworks can be chosen to train a model in the model training step, such as Tensorflow [175], Keras[38], Pytorch [143], etc.

ML models are trained through diverse ML frameworks, thus causing the runtime of the ML inference services to be different. Early works on ML inference systems such as Clipper [28] and Rafiki [191] deployed models in containers using custom runtime and implemented an abstract layer between clients and models to achieve model selection and request batching. Currently, there are several open-sourced runtimes for online ML inference services in production, such as Tensorflow serving [176], TorchServe [144], Kserve [82], or Seldon [161], as well as some optimized libraries for ML, such as Intel Math Kernel Library (MKL) [71]. These runtimes may vary but they contain similar functionalities (*e.g.,* model version management, model warmup) and configuration settings (*e.g.,* parallel threading model, batching, and caching) [176].

The objective is to train and serve ML models into containers and then we could analyze the performance of several schemes to deploy an online ML inference service on a Kubernetes cluster with multi-core machines.

2.3.3 Autonomous Management and Supervision for ML Workflows

Lately, many data researchers and companies have been interested in automating the ML tasks within a training workflow (*e.g.,* AutoML) in order to construct ML models efficiently [199][155][19][38][77]. However, these powerful AutoML modules and frameworks (*e.g.,* Kubeflow Pipelines) are turned off after training a model, thus cannot help the model after being deployed to meet dynamic changes.

To make an autonomic system for ML, Kedziora et al. [78] defined an autonomous system (*i.e.,* AutonoML) as one showing fundamental characteristics of persistence and adaptation. Persistence means that an AutonoML system should be capable of operations in the long term, and adaptation refers to the theories and practices of facing dynamic contexts. Zliobaite et al. [211] identified challenges in designing and building adaptive learning (prediction) systems to achieve scalability, usability, and trust, taking into account various application needs. These works provide a conceptual level view or framework without practical implementation or evaluation.

Seldon[39] provides a set of tools for deploying ML models at scale and presents their practical oversight and governance for ML deployments. But these tools (so-called Alibi)

[36]https://www.kubeflow.org/
[37]https://argoproj.github.io/
[38]https://keras.io/
[39]https://www.seldon.io/

mainly focus on metrics monitoring, outliers and drift detection, and model explanation [80], rather than autonomically taking actions to maintain the model performance. KubeDL[40] supports running different deep learning workloads on Kubernetes. It considers training, model version, model serving, and also an auto-configuration framework Morphling [189] to tune the best configuration before the serving service is deployed. However, the training steps are considered as jobs and the model serving is considered as a simple service rather than a ML workflow, losing the flexibility of using workflows, and also autonomy is not considered. KServe[41], formerly KFServing and used by Kubeflow, enables serverless model inference on Kubernetes. It encapsulates the complexity of autoscaling, networking, health checking, and server configuration to bring serving features like GPU autoscaling, scale to zero, and canary rollouts to the ML deployments. However, it is based on the serverless model supported by Knative, which can only support streaming online inference. Moreover, it integrates Alibi add-ons to detect anomalies, but not deal with them autonomically.

Some adaptive learning algorithms are designed for streaming (unpredicted new data arrives) [42][106][68]. Gama et al. [42] presented a survey on concept drift adaptation, which introduces the online adaptive learning processes and algorithms. Imbrea [68] proposed a framework for implementing AutoML on data stream architectures in production and indicated that, in the presence of concept drift, detection or adaptation techniques have to be applied to maintain predictive accuracy over time. These adaptive learning systems or methods can deal with partially dynamic contexts, but we propose autonomy should be applied at multiple levels to handle both robustness and efficiency problems.

Autonomic computing theories and practices have been used in multiple areas. Formerly, they were applied in the service-oriented computing paradigm [21]. Lately, as systems were adopting the microservice architecture, Our previous work [103] studied the autonomy in those microservice-based systems. Also, some works showed the usage of agents for autonomic computing [30][177][21]. These related works did not directly show how to bring autonomy to ML workflows, but they inspired our work for adopting an agent-based autonomic approach.

Previously, we presented Scanflow, an executor multi-graph framework for end-to-end ML workflow management and debugging in an offline mode, in the form of a proof-of-concept prototype running in a single node, which featured an anomaly detector of out-of-distribution samples in the inference phase [20]. However, to operate models in the long term in an online manner, Scanflow needs to be redesigned from scratch to upgrade the executor nodes to a multi-agent system and to be fully integrated with the resource managing platform to achieve autonomic management and online supervision for the models. On one side, this will help to achieve model scalability, usability, and performance, and, on the other side, it will also contribute to model robustness.

Therefore, in Chapter 5, we investigate a platform (Scanflow-K8s) for autonomic ML workflows with abilities for multi-layered control, based on an agent-based approach that enables autonomic management and supervision of ML workflows at the application layer and the infrastructure layer

[40] https://kubedl.io/

[41] https://kserve.github.io/website

2.3.4 Deployment and Scheduling Schemes for Containerized HPC Workloads

Former works in this area have focused on deploying containerized HPC workloads in traditional HPC systems. These systems have batch-oriented workload managers or resource managers, such as Slurm [167] or Torque, and some of them have included container support [168]. The convergence between HPC systems and Cloud environments has also been explored [210][125], but these works mainly divide the nodes into clusters for different usage and enable access to the HPC cluster from the Cloud cluster.

Beltre et al. [18][158] did some performance analysis on enabling HPC workloads on Cloud infrastructure. They analyzed the HPC workload performance while using different container orchestrators like Kubernetes and Docker Swarm and different networks like InfiniBand. They used the Kubernetes default scheduler. Misale et al. [127] introduced KubeFlux, a Kubernetes plugin scheduler that is based on Flux graph-based scheduler. This plugin translates the Pod into a Flux job and uses the policy within Fluxion to allocate jobs. Saha et al. [157] showed how MPI applications can be scheduled by Mesos using a policy-based approach.

There are also some works focusing on the policies for scheduling HPC jobs in the Cloud. For instance, Gupta et al. [52] presented novel heuristics for online application-aware job scheduling in multi-platform environments. Fu et al. [41] proposed a progress-based container placement for short-lived containerized jobs. Aupy et al. [10] provided an optimal job reservation strategy in scheduling to minimize the cost.

HPC community has important performance considerations on its workloads. Therefore, trialing new deployment schemes for different types of HPC workloads to improve their performance is necessary. Walkup et al. [188] reported best practices for running compute-, memory-, and network-intensive HPC workloads on the Cloud. Medel et al. [118] conducted a performance analysis over Kubernetes considering the deployment and initialization overhead as well as understanding the performance of different Pod settings. Moreover, they provided a rule to decide the number of containers per pod by considering the characteristics of the application. Our performance analysis work in Chapter 3 has demonstrated systematically that i) some types of containerized HPC applications can exploit multi-container deployments which partition the processes that belong to each application into multiple containers in each host in order to achieve better performance; ii) some types of HPC applications gain benefits when using containers by constraining them to a single NUMA domain or pining to specific processors [99][101]. These works show some ways in the Cloud to achieve better performance for HPC workloads, but those insights are not yet being integrated and utilized by the current Cloud schedulers. So, in Chapter 6, our work towards the fine-grained scheduling schemes for HPC workloads on Kubernetes clusters.

Chapter 3

Multi-container Deployment Schemes for HPC Workloads

This chapter is based on the following journal publications:

[**1**] Peini Liu and Jordi Guitart, "Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study", *The Journal of Supercomputing*, vol. 77, no. 6, pp. 6273-6312, June 2021. DOI: 10.1007/s11227-020-03518-1. (JCR Q2).

[**2**] Peini Liu and Jordi Guitart, "Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study", *Cluster Computing*, vol. 25, no. 2, pp. 847-868, April 2022. DOI: 10.1007/s10586-021-03460-8. (JCR Q2).

In this chapter, several performance analyses of multi-container deployment schemes for HPC applications are provided. In Section 3.1, the introduction of multi-container deployment schemes is elaborated. Section 3.2 profiles HPCC benchmarks used in the evaluation. Then, three cases for performance analyses are presented in Sections 3.3-3.5: performance analyses of multi-container deployment and potential use of CPU/Memory affinity on a single-node are described in Section 3.3 and Section 3.4, respectively, and Section 3.5 provides performance analyses of multi-container deployments on an Infiniband cluster and focuses more on different network interconnects and protocols. The conclusions and future work are presented in Section 3.6.

3.1 Introduction

Modern computing infrastructure is evolving at a fast pace from using dedicated physical data centers to cloud computing services. Virtualization, as a fundamental technology for cloud computing, allows efficient utilization and easy maintenance of the infrastructure. So far, this attractive paradigm has been widely used by leading commercial companies and communities to manage their clusters [72][153]. The HPC community is also involved in this transformation of adopting virtualization to benefit from some of its well-known advantages [202], such as the encapsulation of specific software environments for each user, which allows for customization, portability, and research reproducibility [90]; the isolation of users from the underlying system and from other users, which allows for security and fault protection; and the agile and fine-grain resource allocation and balancing, which allows for efficient cluster utilization and failure recovery [37].

Virtualization was initially adopted in the form of hardware virtualization, which adds a layer of software between the operating system and hardware (so-called hypervisor), as well as an extra operating system for the guest. Historically, this incurred noticeable performance penalties, which have been dramatically reduced with the latest advances in virtualization. In particular, HPC workloads have taken advantage of the ability to leverage compute accelerators such as Graphics Processing Units (GPUs) from the virtual machines, or the ability to map the physical resources directly to the virtual machines. Furthermore, innovative deployment schemes have been also proposed to deal with the performance bottlenecks of virtualized HPC workloads in typical HPC architectures, such as multi-socket multi-core systems, like partitioning HPC applications into several virtual machines to prevent them spanning multiple NUMA domains [67]. This allows enabling affinity to a NUMA domain, which can enhance data locality in the L3 cache and reduce the RAM memory latency by preventing access to remote domains. Leveraging processor affinity as well can prevent process preemption and also enhance data locality in the L1 and L2 caches. Apart from exploiting data locality, partitioning can also optimize the packing of virtual machines and hence increase the utilization of the hosts since small-sized tasks can be allocated more easily without blocking in a waiting queue [53]. It can be also helpful to enhance the fault tolerance of the application, by replicating specific vital processes in separate virtual machines. If one of them fails, the replica can take over without downtime. Similarly, specific tasks of the application can also be checkpointed and recovered in case of a failure.

However, the still existing performance degradation of hardware virtualization regarding bare-metal executions [165] might not be acceptable for some HPC users. The emergence of containerization can alleviate that performance gap [37][181], as each container shares the underlying host kernel for OS services such as libraries, modules, and kernel functions. Therefore, a systematic analysis of HPC workloads running on containerized environments is necessary to understand the performance implications of using container technologies for deploying HPC workloads [25][202], and to determine if the partitioning deployment schemes using multiple instances and the performance optimization methods based on affinity used for virtual machines are also appropriate with containers and what potential problems they might incur.

Performance analysis of HPC applications in containerized environments is an ongoing research problem [8][158][196]. Most related works evaluate single-container deployments and emphasize the possibility that deploying a HPC workload into a single container can achieve native performance [37][181]. However, there is a lack of research on multi-container deployment solutions for a single-tenant multi-process HPC workload. Unlike the multi-container deployments holding workloads for multiple tenants [72][74][75][114], using multiple containers to package a single-tenant multi-process/thread HPC workload refers to partitioning the processes or threads belonging to each application into different containers, obtaining in that way a finer-grained deployment. Whereas few works include experiments with different container granularity [25][157][158], none of them provide a deep understanding of the impact of such multi-container deployments on the performance of HPC workloads, which considers different containerization technologies, container grain sizes, and hardware platforms. To better identify optimal containerized deployment schemes and potential performance bottlenecks in a single multi-socket multi-core system, a proper and in-depth performance analysis is important before migrating HPC applications to containerized environments.

In addition to the above context, a matter of the utmost importance for HPC users is that the containers running their applications can leverage the underlying HPC resources such as Infiniband networks, which offer high-speed networking capabilities with improved throughput and low latency through the use of Remote Direct Memory Access (RDMA) [204].

Previous work has demonstrated that containerized HPC applications can exploit InfiniBand networks, especially when they run on a single container per host that shares the host network namespace. The ability to provision InfiniBand to Docker and Singularity containers has been shown in [158][24][204][206]. Whereas some works have evaluated more sophisticated networking modes, such as overlay networks, they have just superficially considered multi-container deployments which partition the processes that belong to each application into multiple containers in each host. Partitioning HPC applications has been demonstrated to be useful when using virtual machines by constraining them to a single NUMA (Non-Uniform Memory Access) domain [67], and can also increase the utilization of the hosts since small-sized tasks can be packed more easily. However, it is still unclear how multi-container deployment schemes with different affinity settings perform with various network interconnects and protocols, and how different communication patterns and message sizes impact the performance of containerized HPC workloads. Consequently, it is essential to understand the performance implications of multi-container deployment schemes for HPC workloads on Infiniband clusters, focusing especially on understanding how the container granularity and its combination with processor and memory affinity impact the performance when using different networking modes.

Early containerization implementations for deploying HPC benchmarks were mainly Linux-VServer, OpenVZ, and LXC [196]. However, containerization technologies have been evolving and Docker[1] has become the most popular containerization software [11][25][158]. Docker provides an easy way to isolate the network and limit the resource usage of the containers, but some challenges remain with this technology to guarantee security and ensure performance when employed in HPC. Singularity[2], a novel HPC-oriented containerization technology, offers promising solutions for these issues [8][91]. Regarding security, Singularity does not create containers as spawned child processes of a root owned daemon. Regarding performance, Singularity enables all the containers to use the underlying HPC environment in a natural way (without namespaces isolation). This work focuses on these two containerization technologies for deploying HPC workloads.

This chapter presents a systematic performance comparison and analysis of containerized deployment schemes for HPC workloads. We address the next research questions:

1. What is the impact of container granularity on the performance of multi-container deployment schemes for HPC workloads?
2. What is the impact of processor and memory affinity on the performance of multi-container deployment schemes for HPC workloads?
3. What is the impact of container technologies, container granularity, processor and memory affinity on the performance of multi-container deployment scenarios using different network interconnects and protocols?

3.2 Profiling Analysis of the HPCC Benchmarks

Due to the different attributes of each benchmark in HPCC benchmark suite, some profiling of these benchmarks is useful for understanding their different MPI usage patterns [62], and can be used as a baseline for comparison with container-based executions in the evaluation.

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

Our analysis considers two different application subscription patterns, namely exactly-subscribed mode and over-subscribed mode. In the exactly-subscribed mode, the number of running processes is equal to the number of available processors. In the over-subscribed mode, there are more processes running than processors available, that is, it permits resource over-subscription. Tasks enabling over-subscription can obtain their resources sooner and decrease the waiting time, thus can be started earlier than in the exclusive mode [173].

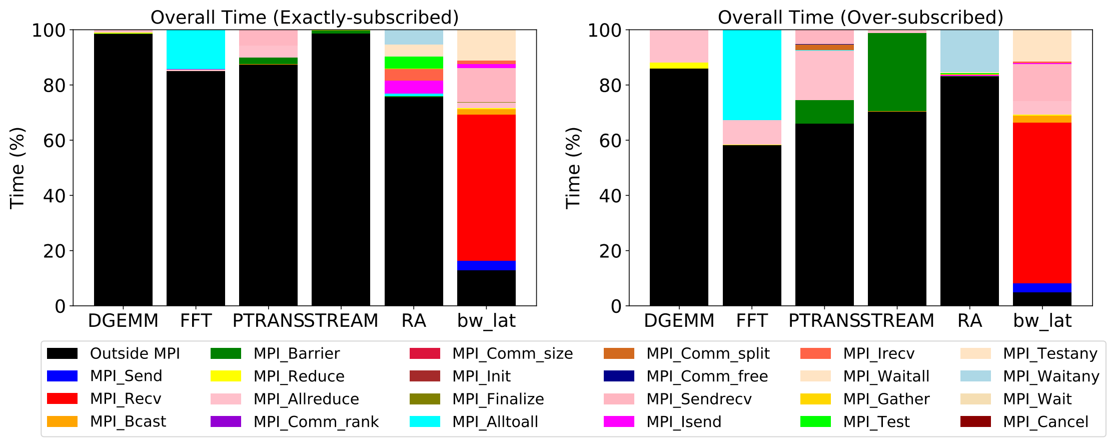


Figure 3.1: HPCC MPI profiling analysis.

Environment and Settings: The hardware platform consists of a single host with 2 x Intel 2697v4 CPUs (18 cores each, hyperthreading disabled), 256 GB RAM, 60 TB GPFS file system, and 1 Gb Ethernet network. The operating system on each server is CentOS 7.6. OpenMPI v4.0.3rc3 and HPCC benchmarks v1.5.0 are compiled by the GNU compiler collection in version 5.5.0. All the benchmarks are running with 16 processes on bare-metal. Exactly-subscribed mode is using 16 cores (8 from each socket). For over-subscribed mode, the over-commitment ratio is set to 2, which means using 8 cores (4 cores from each socket). We use an open source analysis tool Paraver[3] to profile MPI usage patterns of the benchmarks [142].

Results: Figure 3.1 shows the HPCC MPI profiling results. Segments of different colors correspond to the time spent within the various MPI functions with respect to the overall execution time. Table 3.1 presents the detailed time consumption percentages and the number of invocations of these MPI functions, which are classified according to the corresponding communication patterns.

From these results, we can divide these benchmarks broadly into two categories: MPI communication workloads, where processes need to communicate (frequently) with each other, and MPI throughput workloads, where there is (almost) no communication between processes [185]. Benchmarks whose name starts with G- (G stands for Global) and b_eff benchmark belong to the first category while other benchmark names starting with EP- (EP stands for Embarrassingly Parallel) belong to the second one. Within the MPI communication workloads, b_eff presents different patterns of point-to-point communications (*e.g.,* blocking ping-pong transfer, blocking concurrent transfer, and non-blocking communications), which are also shown in G-PTRANS (blocking concurrent transfer) and G-RandomAccess (non-blocking communications). G-FFT uses mainly collective

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

Table 3.1: HPCC benchmark profiling analysis for Exactly- and Over-subscribed mode.

Benchmark		Point to point communications				Collective communications				
		blocking ping-pong transfer	blocking concurrent transfer	non-blocking transfer	non-blocking synchronize	barrier synchronize	data movement			global reduce
		MPI_Send MPI_Recv	MPI _Sendrecv	MPI_Isend MPI_Irecv	MPI_Wait (any/all) MPI_Test (any/all) MPI_Cancel	MPI_Barrier	MPI _Alltoall	MPI _Bcast	MPI _Gather	MPI_(All) Reduce
DGEMM		E[1]:<0.01% O[1]:<0.01% NI[2]:210				E:<0.01% O:<0.01% NI:16		E:0.01% O:0.01% NI:96		E:1.41% O:13.88% NI:96
FFT		E:<0.01% O:<0.01% NI:210				E:<0.01% O:0.02% NI:16	**E:14.22% O:32.65% NI:96**	E:<0.01% O:0.01% NI:96		E:0.56% O:9% NI:48
PTRANS		E:<0.01% O:<0.01% NI:220	**E:5.69% O:5.24% NI:480**			E:2.52% O:8.43% NI:96		E:<0.01% O:<0.01% NI:80		E:4.26% O:18.09% NI:592
STREAM		E:<0.01% O:<0.01% NI:210				E:1.25% O:28.42% NI:1296		E:<0.01% O:<0.01% NI:144	E:0.05% O:0.13% NI:16	E:0.05% O:0.92% NI:256
RA		E:<0.01% O:<0.01% NI:210		**E:8.89% O:0.62% NI:3276632**	**E:14.08% O:16.07% NI:3276344**	E:0.42% O:0.03% NI:4096	E:0.99% O:0.06% NI:4048	E:<0.01% O:<0.01% NI:96		E:<0.01% O:<0.01% NI:304
b_ eff	ping-pong	**E:56.32% O:61.6% NI:54404**				E:<0.01% O:0.01% NI:16		E:2.1% O:2.53% NI:6672		E:2.36% O:5.22% NI:9504
	ring		**E:12.41% O:13.33% NI:50784**	**E:2.67% O:0.91% NI:101568**	**E:11.11% O:11.48% NI:25392**					

[1] *(E) and (O) means Exactly- and Over-subscribed Mode, respectively.*
[2] *NI means Number of Invocations.*

all-to-all communication. Thereby, all of these benchmarks can be used to evaluate the different aspects of interprocess communication. On the other side, MPI throughput workloads EP-STREAM and EP-DGEMM can be used to assess the memory bandwidth and the computation performance of the system, respectively. Note that our classification matches with the existing literature on HPCC [113], which has identified b_eff, G-RandomAccess, G-PTRANS, and G-FFT as performance-sensitive to the interconnection latency and/or bandwidth, whereas EP-DGEMM and EP-STREAM have been characterized as not sensitive to them.

3.3 Performance Analysis of Multi-container Deployments

This section addresses the first question: What is the impact of container granularity on the performance of multi-container deployment schemes for HPC workloads?

3.3.1 Objective

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

3.3.2 Method

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

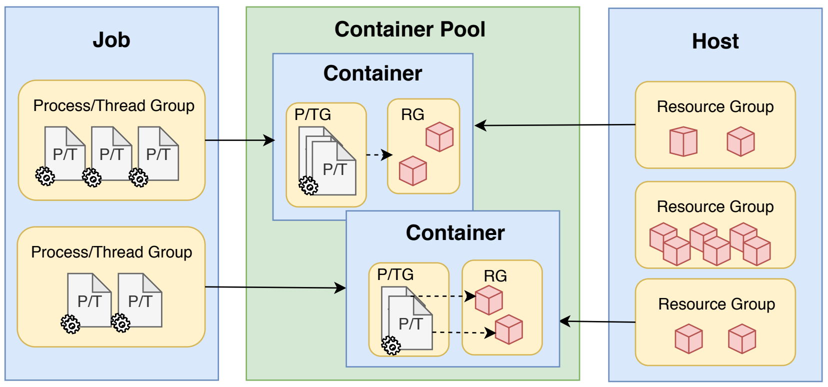


Figure 3.2: Container-based deployment model for HPC workloads.

The container-based deployment model for HPC workloads is shown in Figure 3.2. It consists of three modules: a job contains several processes/threads, which can be divided into groups of various sizes and are packaged into different containers; a host has multiple resources organized into sets which are able to run the containers; and a containerization layer including containers that holds the mapping between a group of processes/threads and a set of resources. Our evaluation compares and analyzes the performance of HPC workloads by considering deployment schemes with different number of containers and processes per container and using different containerization technologies on the containerized layer.

3.3.3 Experimental Setup

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

Classical unpaired two-samples T tests require that the two groups of samples are normally distributed, so we first verify that by using Shapiro-Wilk tests [164]. When some of the groups of samples being compared are not normally distributed, we use Mann-Whitney tests [115] instead of the classical two-samples T tests. Unpaired two-samples T tests also require that the variances of the two groups are equal. We verify this by using Fisher's F-tests. When the variances are not equal, we use Welch T tests [193] instead of the classical T tests.

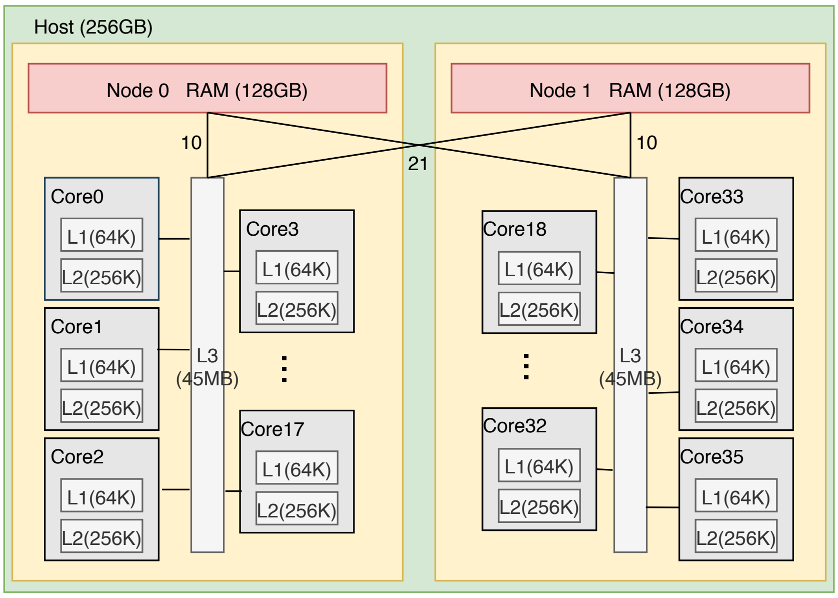


Figure 3.3: A schematic view of our single-host HPC platform with two sockets and 18 cores per socket with shared L3 cache.

Environment: Our experiments are executed on a single-host HPC platform. The hardware characteristics of this host have been described in Section 3.2. Figure 3.3 shows

Table 3.2: Overview of the multi-socket multi-core hardware settings used in the experiments.

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

Table 3.3: Software stack.

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

a schematic view of its architecture. There are two sockets containing 18 cores each. The distance for accessing local and remote memory is 10 and 21, respectively. Each core has its own L1 and L2 cache, and L3 cache is shared by the 18 cores in the same socket. We use this host to define two different hardware platform settings: one with Non-Uniform Memory Access (NUMA) and another with Uniform Memory Access (UMA). Table 3.2 summarizes the hardware characteristics of these two settings. Both of them have the same number of cores, each one with L1 data cache (32K), L1 instruction cache (32K), and L2 cache (256K). In the NUMA hardware setting, those cores belong to 2 different sockets, each one with its own L3 cache (45MB); in the UMA hardware setting, the cores all belong to a single socket with a single 45MB L3 cache. The software stack for both host and containers, and its compilation environment are described in Table 3.3.

Benchmark settings: The settings for HPCC are the same as described in Section 3.2, so all the benchmarks are running with 16 MPI processes in all the scenarios. In the exactly-subscribed mode, those processes run on 16 cores, whereas in the over-subscribed mode they run on 8 cores. Additionally, we enable OpenMPI MCA parameter `mpi_yield_when_idle` for all the over-subscribed scenarios to prevent the degradation from the OpenMPI (see Section 3.3.4).

Container granularity settings: Different deployment schemes for evaluating container granularity are presented in Figure 3.4. Figure 3.4 (a) presents the settings of deployment scenarios on the NUMA hardware platform setting and Figure 3.4 (b) on the UMA hardware platform setting. E or O refers to the application running on exactly-subscribed mode or over-subscribed mode, respectively. E1-E5 and O1-O4 reflect the different granularity of the containers. As shown in Table 3.8, E1 and O1 use a single container, while E2-E5, O2-O4 are scenarios with an increasing number of containers, but decreasing number of processes per container.

In the experiments denoted as 'ANY', each container could use any of the available cores according to the used hardware platform setting (see Table 3.8). The actual distribution of the running processes on the available cores is decided dynamically by the CFS Linux scheduler. In the experiments denoted as 'PIN', we enforce a 1-to-1 binding from the processes of the application to the available cores according to the used hardware platform settings. This binding holds during the entire execution of the application. We include this setting in the comparison to serve as a reference and to assess the performance reproducibility when variable process placement is eliminated,

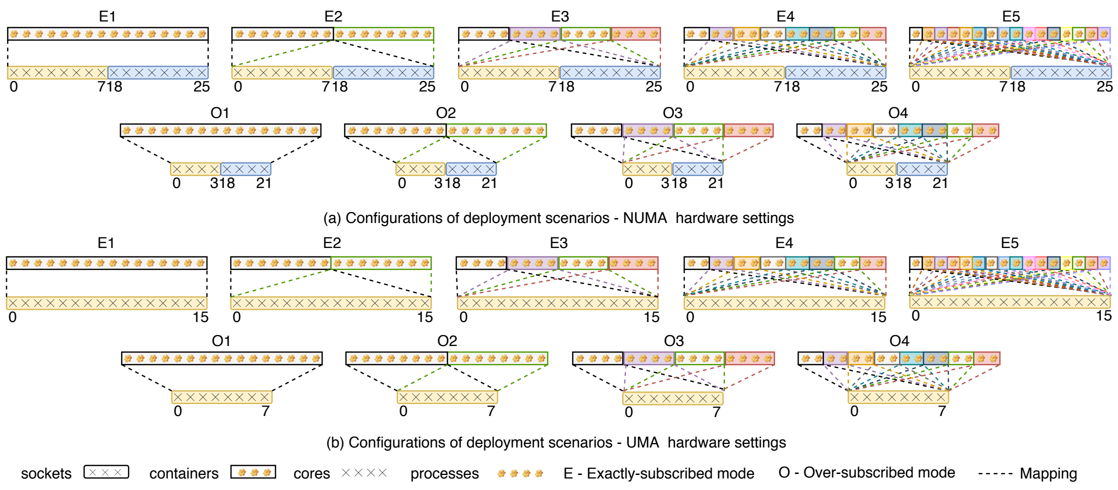

E1
E2
E3
E4
E5
O1
O2
O3
O4
0
7
18
25
3
21
15
(a) Configurations of deployment scenarios - NUMA hardware settings
(b) Configurations of deployment scenarios - UMA hardware settings
sockets
containers
cores
processes
E - Exactly-subscribed mode
O - Over-subscribed mode
Mapping

Figure 3.4: Containerized deployment scenarios.

Table 3.4: Settings for containerized deployment scenarios.

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

especially with over-subscription.

Containerization technologies: Docker and Singularity containerization technologies are evaluated in this work. We include also some variants of Singularity in the comparison. The different features of these technologies are described in Table 3.5: 1) *Docker*: Docker containers run isolated into different namespaces and cgroups. 2) *Singularity*: Default Singularity version, which executes the containers like they were native programs or scripts on a host computer, without encapsulating them on separated namespaces or cgroups. 3) *Singularity-instance*: Similar to Docker, Singularity container instances, which are persistent versions of the container image, run isolated in the background into different namespaces and cgroups. 4) *Singularity+cgroup*: Plain Singularity containers are executed (not instances), but each of them runs on its own cpu cgroup. Note that this cgroup should be a hierarchical cgroup, otherwise the scheduler will allocate resources among multiple root level cgroups thus bringing some degradation of performance.

Performance analysis tools: We use Paraver to profile MPI usage patterns of the benchmarks. We capture also performance event counters (through Perf[4]) and operating system metrics, such as context-switches, migrations, and memory accesses, from representative executions of the benchmarks and we use them to explain the obtained performance results.

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

Table 3.5: Features of different containerization technologies.

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

3.3.4 Results

① · Impact of containerization technology and container granularity on multi-container deployments

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

MPI communication workloads: As a result of the profiling analysis in Section 3.2, we concluded that b_eff (including Ping-Pong and Ring patterns), G-RandomAccess, G-PTRANS, and G-FFT benchmarks can be classified as MPI communication workloads. In particular, b_eff spends about 87% of overall runtime in MPI (95% when over-subscribed), G-RandomAccess spends about 24% of overall runtime in MPI (16% when over-subscribed), G-PTRANS spends about 13% of overall runtime in MPI (34% when over-subscribed), and G-FFT spends about 15% of overall runtime in MPI (42% when over-subscribed).

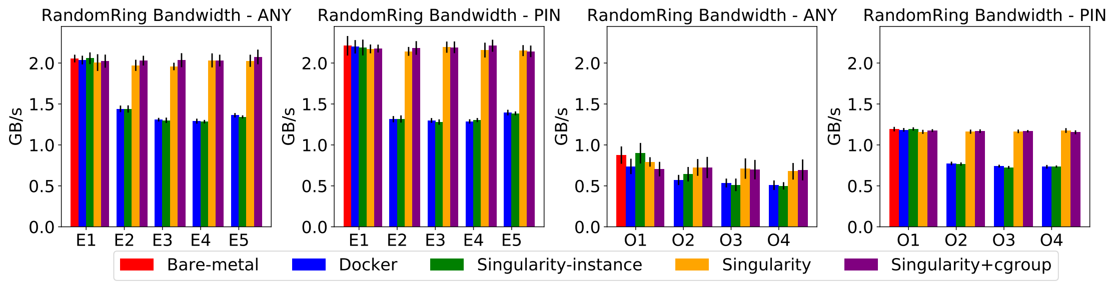


Figure 3.5: Impact of container granularity in b_eff(RandomRing) bandwidth on NUMA hardware platform setting.

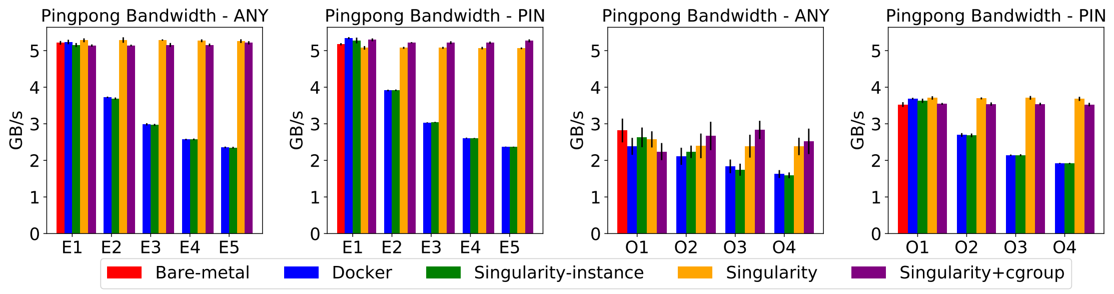


Figure 3.6: Impact of container granularity in b_eff(PingPong) bandwidth on NUMA hardware platform setting.

Figures 3.5 and 3.6 show the bandwidth results for the b_eff benchmark, differentiating PingPong and RandomRing MPI point-to-point communication patterns. We omitted the latency results as they were essentially following the same trend. There is significant performance degradation when the processes run on multiple containers in Docker and Singularity-instance (scenarios E2 to E5 and O2 to O4) regarding single-container deployments, as the P-values of the corresponding T-tests range from $4.9e^{-21}$ to $3.3e^{-11}$, which are all clearly lower than 0.05. In order to better understand this behavior, Figures 3.7 and 3.8 detail the time spent in seconds on each MPI function for the various communication patterns in this benchmark when running on Docker. This time is greater when running with multiple containers for blocking ping-pong transfer patterns (MPI_Send and MPI_Recv), around 90% on scenario E2, and non-blocking transfer patterns (MPI_Isend and MPI_Irecv), around 37% on scenario E2, and increases with the number of containers, +16% (E3), +7% (E4), +4% (E5) and +12% (E3), +8% (E4), +4% (E5), respectively. The time is also greater when running multiple containers for non-blocking synchronization, around 47% on scenario E2, and blocking concurrent transfer patterns, around 61% on scenario E2, but it barely increases with the number of containers, +7% (E3), +5% (E4), -3% (E5) and +4% (E3), +1% (E4), -4% (E5), respectively.

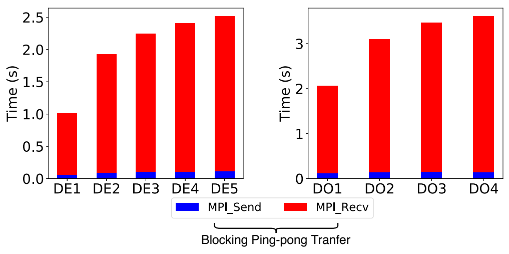


Figure 3.7: Time spent in MPI communication patterns of b_eff(PingPong) benchmark for PIN scenarios on Docker (NUMA hardware platform setting).

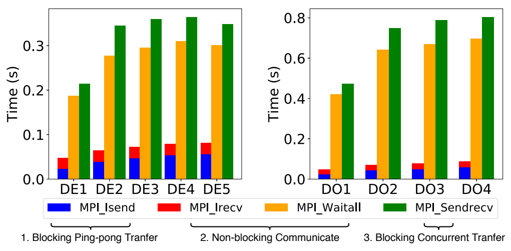


Figure 3.8: Time spent in MPI communication patterns of b_eff(Ring) benchmark for PIN scenarios on Docker (NUMA hardware platform setting).

This degradation occurs because the processes running on separated containers in Docker and Singularity-instance are deployed on isolated network namespaces and have to use the TCP/IP network stack rather than shared-memory to communicate with one another. Executions with a single container or using Singularity do not have degradation on any of the scenarios when comparing with bare-metal (the P-values of the corresponding T-tests range from 0.34 to 0.89, clearly higher than 0.05, thus the difference is not statistically significant) because the processes do not communicate through isolated network namespaces. All the processes belong to the same namespace and can use shared-memory to communicate as when running on bare-metal.

According to the profiling analysis of benchmarks in Section 3.2, among the benchmarks classified as MPI communication workloads, G-RandomAccess and G-PTRANS present some MPI point-to-point communication patterns (see Figures 3.9-3.10). In particular, G-RandomAccess spends about 23% of overall runtime on point-to-point non-blocking communication (16% when over-subscribed). Thereby, Docker and Singularity-

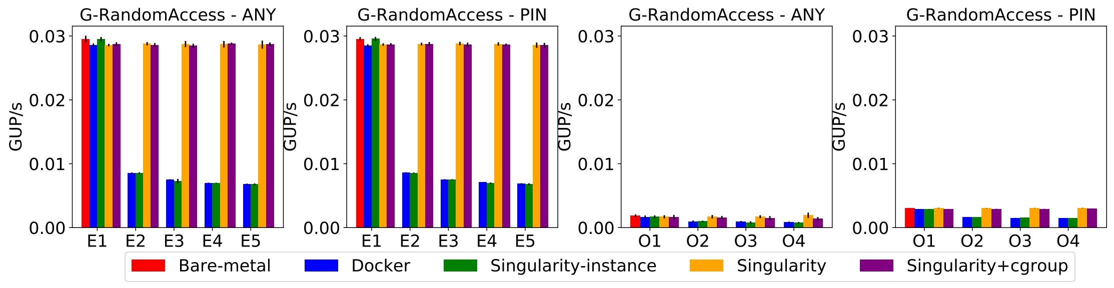


Figure 3.9: Impact of container granularity in G-RandomAccess performance on NUMA hardware platform setting.

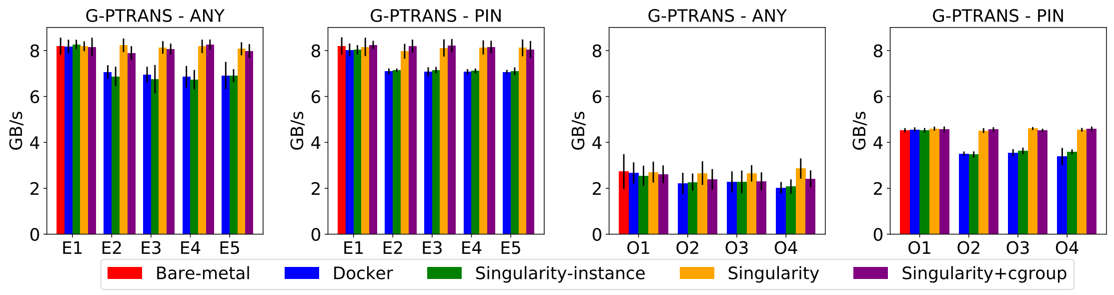


Figure 3.10: Impact of container granularity in G-PTRANS performance on NUMA hardware platform setting.

instance incur performance degradation (around 70% on scenario E2) that increases slightly with the number of containers (up to 77% on scenario E5). This degradation is statistically significant as the P-values of the corresponding T-tests are lower than 0.05 (ranging from $9.4e^{-22}$ to $1.8e^{-18}$). G-PTRANS spends about 5% of overall runtime on point-to-point blocking concurrent transfers (*e.g.,* MPI_Sendrecv) (also 5% when over-subscribed), thus Docker and Singularity-instance degrade on running multiple containers due to using the network stack (around 15%-17% degradation on scenarios E2-E5, which is statistically significant as the P-values of the T-tests with respect to single-container deployments range from $3.2e^{-7}$ to $1.8e^{-4}$, which are lower than 0.05), but this degradation does not increase with the number of containers (P-values of scenarios E3-E5 with respect to E2 range from 0.2 to 1, which are higher than 0.05). The performance degradation in G-PTRANS is significantly lower than b_eff and G-RandomAccess because the number of invocations to MPI functions is considerably lower. As before, Singularity and Singularity+cgroup do not incur any statistically significant degradation. This is confirmed in the corresponding T-tests where all the P-values are higher than 0.05 (ranging from 0.07 to 0.9).

Unlike previous MPI communication workloads, G-FFT mainly uses collective communication (mostly MPI_Alltoall) for data movement. From the results in Figure 3.11, the performance degradation when running multiple containers in Docker and Singularity-instance is almost negligible when exactly-subscribed (around 1%) and quite small when over-subscribed (around 4% when pinning processes). In both cases, the P-values of the corresponding T-tests show that those small differences are statistically significant and not due to randomness (ranging from $5.3e^{-6}$ to $7.5e^{-3}$). What makes the difference is the number of invocations of MPI calls. As a rule of thumb, applications doing

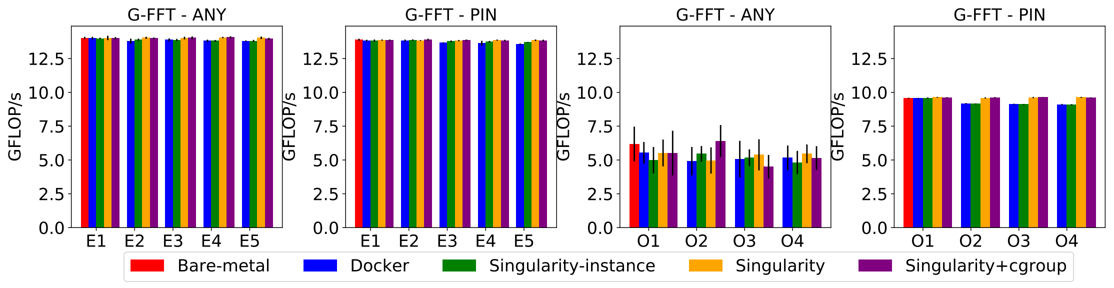


Figure 3.11: Impact of container granularity in G-FFT performance on NUMA hardware platform setting.

point-to-point communication perform many more invocations to MPI functions than applications using collective communication. In particular, as shown in Table 3.1, G-FFT does only 210 point-to-point and 256 collective invocations (vs. more than 50000 point-to-point invocations in b_eff(PingPong)), hence the degradation is considerably lower.

MPI throughput workloads: The results of the profiling analysis in Section 3.2 allowed to classify EP-STREAM and EP-DGEMM benchmarks as MPI throughput workloads. As shown in Figures 3.12 and 3.13, which depict the performance of those benchmarks when running with various container grain sizes and containerization technologies, those workloads do not show significant performance variation regarding the baseline when increasing the number of containers per host. For instance, the P-values of the T-tests for multi-container deployments of EP-STREAM regarding single-container deployments range from 0.05 to 0.85 (higher than 0.05). This is due to the low amount of interprocess communication, namely 1.4% of overall runtime in MPI (29.5% when over-subscribed, but mostly in synchronization functions) for EP-STREAM, and 1.5% of overall runtime in MPI (13.9% when over-subscribed, but mostly in the global reduce) for EP-DGEMM.

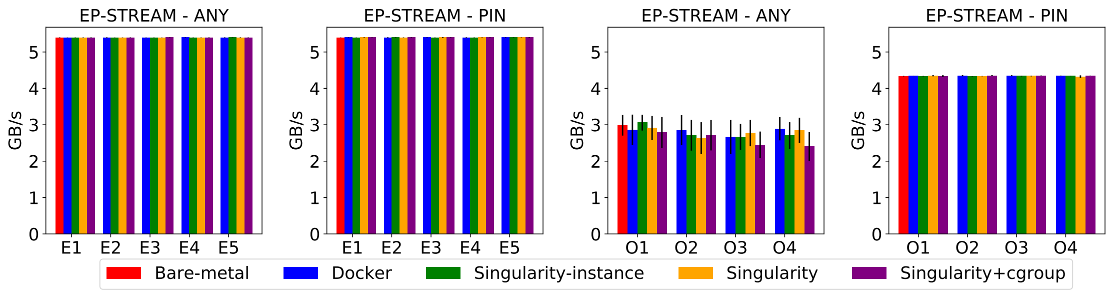


Figure 3.12: Impact of container granularity in EP-STREAM performance on NUMA hardware platform setting.

Something noticeable in Figure 3.13 is the performance improvement of EP-DGEMM in scenario ANY-E5 (which runs a single MPI process on each container) regarding the other deployment scenarios with all the containerization technologies but plain Singularity. In those technologies, ANY-E5 shows significant difference compared to ANY-E1, as the P-values of the corresponding T-tests range from $1.7e^{-3}$ to $7.3e^{-3}$ (lower than 0.05). As shown in Figure 3.14, which depicts relevant performance counters of EP-DGEMM,

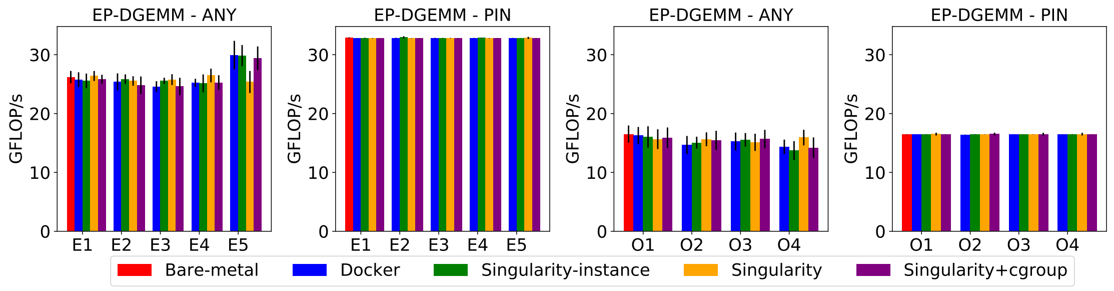


Figure 3.13: Impact of container granularity in EP-DGEMM performance on NUMA hardware platform setting.

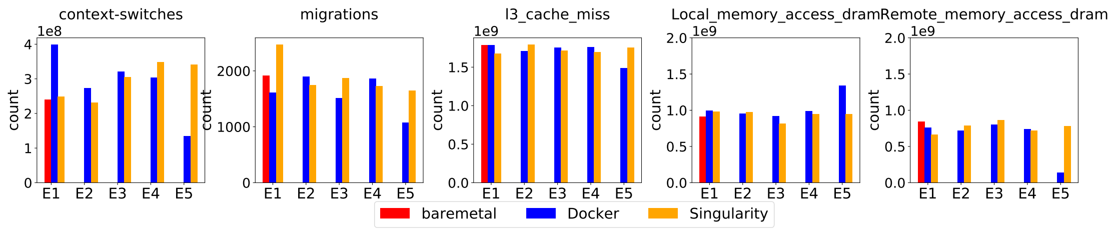


Figure 3.14: Performance event counters of EP-DGEMM for ANY scenarios on NUMA hardware platform setting.

scenario E5 with Docker has considerably less process migrations and context-switches than the other deployment scenarios. It also shows better cache utilization (less L3 misses), more local memory accesses, and only minimal remote memory accesses. These are consequences of the scheduling of the containers (*i.e.,* the cgroups) and their corresponding MPI processes. As each container runs a single process, this is essentially a single-level scheduling (*i.e.,* at the cgroup level), which is simpler and allows to exploit processor affinity better, in a similar way to when processes are pinned explicitly (although not so deterministic). The same occurs with Singularity-instance, but not with Singularity because all the processes run within the same cgroup.

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

② · Impact of the cgroup scheduling on multi-container deployments

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

Linux cgroups are mechanisms from kernel-level that control the resource allocation by restricting CPU, memory, network, etc., for each group of processes. One of them is the CPU controller, which is responsible for grouping tasks together that will be viewed by the scheduler as a single unit. The CFS (Completely Fair Scheduler) scheduler applies the principle of sharing the resources fairly among these groups at the same level of the hierarchy, which means it will first divide CPU time equally between all entities in the same level, and then proceed by doing the same in the next level [46].

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

As shown in previous figures in experiment 1, Singularity+cgroup achieves the same performance as Singularity for all the benchmarks in exactly-subscribed scenarios, but it incurs some performance degradation (similar to Docker and Singularity-instance) in some of the benchmarks on over-subscribed scenarios. For instance, this is especially noticeable with G-PTRANS on ANY scenarios O2, O3, and O4, and EP-DGEMM (and to a lesser extent on G-FFT) on ANY scenario O4. In those scenarios, the cgroup scheduling performed by the CFS results in imbalanced executions. CFS tries to maintain fair time allocation among cgroups, not processes, but it is not especially accurate in tracking the load of scheduling entities when they are groups of processes (*i.e.,* cgroups). Those coarse-grain load measurements are then used to calculate the load of the processors and decide about load balancing from busier to idler processors, resulting in an imbalanced allocation of processes to processors [13][105]. This is critical in over-subscribed scenarios where processes must share processors efficiently to ensure progress.

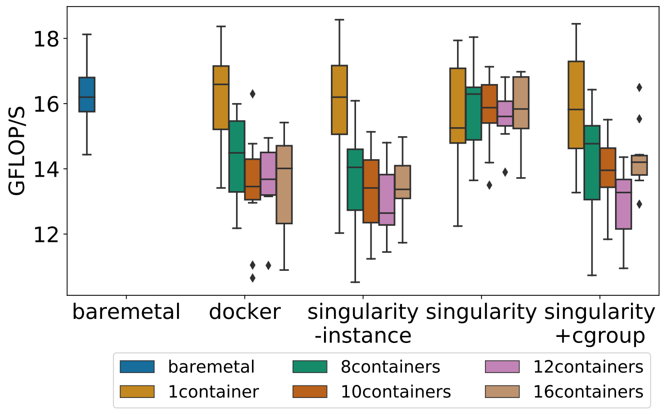


Figure 3.15: Performance comparison of EP-DGEMM with different number of containers.

This can be confirmed in Figure 3.15, which shows the EP-DGEMM performance on over-subscribed mode including additional scenarios that deploy a higher number of containers cgroups than the number of available CPUs. Whereas holding all the MPI processes in a single container provides bare-metal performance, having multi-container deployments causes significant performance degradation in all the containerization technologies except Singularity, as Singularity is not using distinct cgroups. The corresponding T-tests confirm that the P-values for Singularity are higher than 0.05 (ranging from 0.13 to 0.49), whereas for the other containerization technologies they are lower than

0.05 (ranging from $1.9e^{-6}$ to $7.4e^{-3}$).

③ · Impact of the hardware platform setting on multi-container deployments

The hardware platform setting also has an impact on the performance of the different benchmarks, but this is mostly unrelated to the containerization technology and the deployment scheme. As such, in the UMA setting, there is also significant performance degradation for MPI communication workloads when the processes run on multiple containers in Docker and Singularity-instance because the processes running on separated containers are deployed on isolated network namespaces. Executions with a single container, using Singularity, or for MPI throughput workloads do not have degradation on any of the scenarios when compared with bare-metal.

The performance difference between the NUMA and UMA hardware platform settings depends on the specific characteristics of each benchmark. Figures 3.16-3.19, 3.21, and 3.23-3.26 present the performance difference (in %) of UMA relative to NUMA for each benchmark. The difference between these two hardware settings is that UMA optimizes the latency of accessing memory by improving the cache usage and eliminating the remote memory accesses, while NUMA optimizes the memory bandwidth. Given the performance variability in the ANY over-subscribed scenarios, which makes it difficult to obtain meaningful conclusions, we focus the comparison in this section on the exactly-subscribed scenarios.

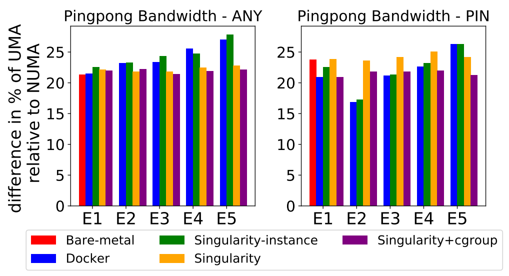


Figure 3.16: Bandwidth difference (in %) of UMA relative to NUMA in b_eff(PingPong).

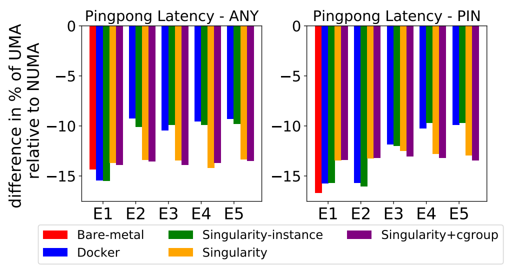


Figure 3.17: Latency difference (in %) of UMA relative to NUMA on b_eff(PingPong).

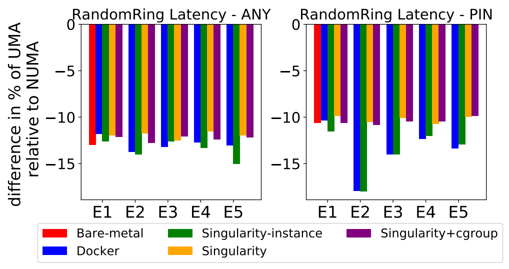


Figure 3.18: Latency difference (in %) of UMA relative to NUMA on b_eff(RandomRing).

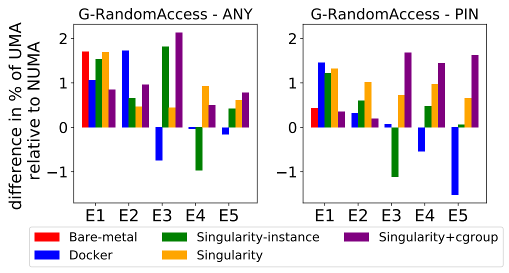


Figure 3.19: Performance difference (in %) of UMA relative to NUMA on G-RandomAccess.

As shown in Figures 3.16-3.19, PingPong Bandwidth/Latency and Ring Latency benchmarks show significantly better performance in the UMA setting, ranging from 15% to 27%. In those benchmarks, the P-values of the T-tests comparing UMA and NUMA scenarios range from $1.0e^{-25}$ to $1.8e^{-4}$ (lower than 0.05). G-RandomAccess benchmark also shows some improvement (less than 2%), but the results are inconclusive, as the P-values of the T-tests are higher than 0.05 for some scenarios (ranging from 0.06 to 0.94) and lower than 0.05 for others (ranging from $2.5e^{-3}$ to 0.04). The MPI processes on these benchmarks communicate through small-sized point-to-point messages and are not memory-intensive. In the UMA setting, all the processes run in a single socket, sharing the L3 cache and the local memory, which enhances the use of the cache (less L3 misses as shown in Figure 3.20) and reduces the number of memory accesses regarding the NUMA setting.

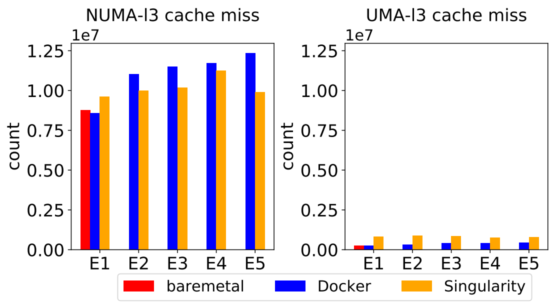


Figure 3.20: Performance counters for b_eff(PingPong) benchmark on ANY scenarios.

As shown in Figure 3.21, EP-DGEMM also performs better in the UMA setting for ANY scenarios. The improvement is significant as the P-values of the corresponding T-tests for scenarios E1-E4 and E5-Singularity range from $1.6e^{-9}$ to $4.5e^{-2}$ (lower than 0.05). The difference is less statistically significant in scenario E5 with the other containerization technologies as the corresponding P-values are around 0.14. Placing all the MPI processes in the same socket has resulted in better cache sharing and allows them to better access the local memory, which reduces the latency of accessing remote memory.

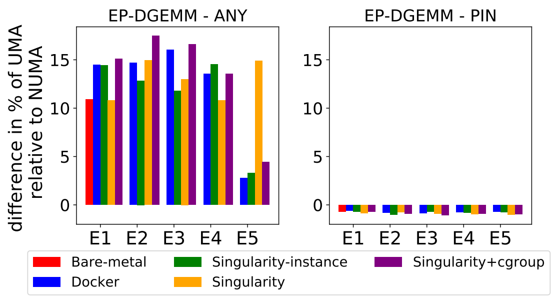


Figure 3.21: Performance difference (in %) of UMA relative to NUMA on EP-DGEMM.

As shown in Figure 3.22, EP-DGEMM in the UMA setting performs only local memory accesses, whereas it does a mixture of local (56% of the L3 cache misses count) and remote memory accesses in the NUMA setting. For PIN scenarios, EP-DGEMM in the NUMA setting already has good memory locality (local memory accesses count is 99% of L3 cache misses count), thus UMA does not bring any advantage on avoiding remote memory accesses latency, and hence the performance of EP-DGEMM on both hardware platform settings are almost the same, with NUMA bringing a small improvement around 1%-2%, which is statistically significant as the corresponding P-values range from $2.0e^{-15}$ to $2.8e^{-3}$.

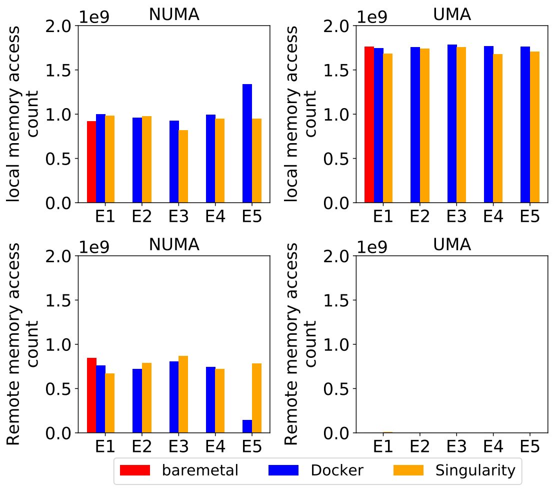


Figure 3.22: Performance counters for EP-DGEMM benchmark on ANY scenarios.

As shown in Figures 3.23-3.26, EP-STREAM, G-FFT, b_eff(Ring Bandwidth), and G-PTRANS have significantly worse performance in the UMA setting. In particular, EP-STREAM is 48% slower in the UMA setting, with P-values of the T-tests ranging from $3.3e^{-36}$ to $1.8e^{-4}$ (lower than 0.05). As the processes are mostly accessing the local memory in the NUMA setting (99.6% for ANY scenarios and 99.9% for PIN scenarios), UMA cannot bring additional benefit by avoiding remote memory accesses, but introduces more memory contention because it has only one socket which reduces the available memory bandwidth. G-FFT, b_eff(Ring Bandwidth), and G-PTRANS communicate their processes using large messages. In a single host, their performance is also limited by the memory bandwidth, and for this reason, the NUMA setting provides better performance for them. For example, G-FFT is 32%-36% faster (P-values ranging from $7.5e^{-35}$ to $1.8e^{-4}$) and G-PTRANS is 14%-20% faster (P-values ranging from $1.0e^{-16}$-$2.8e^{-3}$).

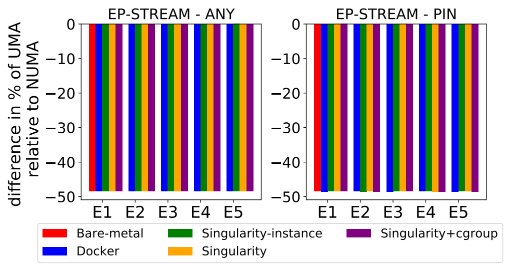


Figure 3.23: Performance difference (in %) of UMA relative to NUMA on EP-STREAM.

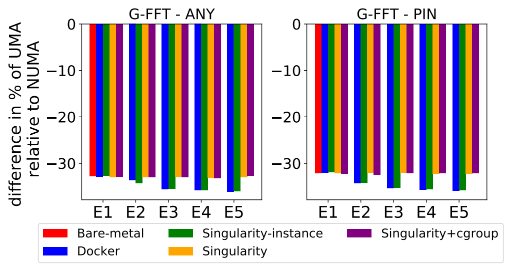


Figure 3.24: Performance difference (in %) of UMA relative to NUMA on G-FFT.

In order to measure the memory contention that occurs on those benchmarks, we

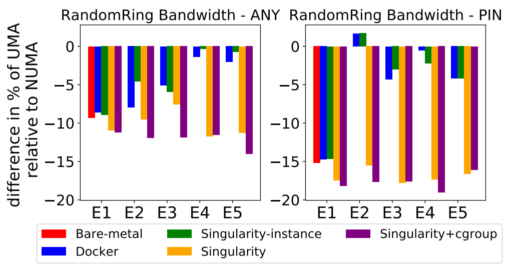


Figure 3.25: Bandwidth difference (in %) of UMA relative to NUMA on b_eff(RandomRing).

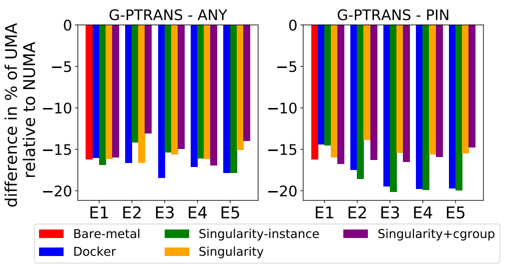


Figure 3.26: Performance difference (in %) of UMA relative to NUMA on G-PTRANS.

calculate the memory contention ratio among cores in the UMA and NUMA settings by using the model proposed by Tudor and Teo [182]. Same as those authors, we are not interested in the absolute value of stall cycles, but on how stall cycles grow relative to a baseline value on one core (where there is no contention) due to memory contention among cores. Consequently, we derive the memory contention ratio ω as the stall cycles due to contention divided by the useful work cycles (including stall cycles that are not due to resource contention). A higher ω means more memory contention. As shown in Figures 3.27 and 3.28, which depict the memory contention ratio for EP-STREAM and G-FFT, respectively, memory contention is higher in the UMA platform setting, because there are 16 processes concurrently accessing the local memory and the UMA platform setting cannot benefit from a second memory controller to serve their operations, which increases the contention in L3 cache and local memory.

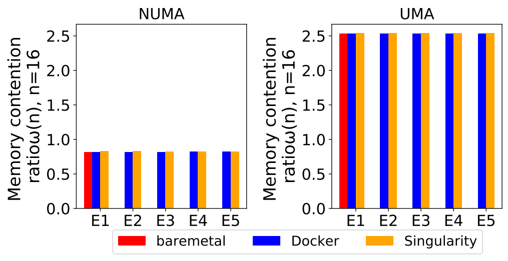


Figure 3.27: Memory contention ratio for EP-STREAM benchmark on ANY scenarios.

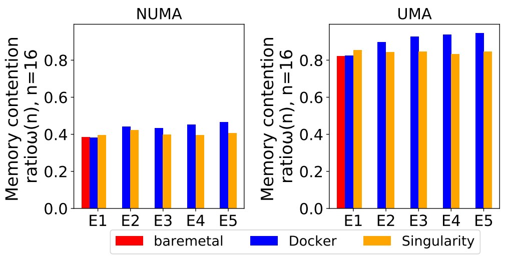


Figure 3.28: Memory contention ratio for G-FFT benchmark on ANY scenarios.

④ · Proper configuration of multi-container deployments with over-subscription

Unlike the exactly-subscribed mode where OpenMPI can run its message passing engine always in aggressive mode (never giving up the processors to other processes), over-subscribed mode requires the OpenMPI engine to run in degraded mode and frequently yielding the processor to its peers when idle, thereby allowing all processes to make progresses [136]. The awareness of the aggressive or degraded mode of OpenMPI engine is usually automatic, although the user can use the MCA parameter `mpi_yield_when_idle` to control whether an MPI process runs in aggressive or degraded performance mode [135].

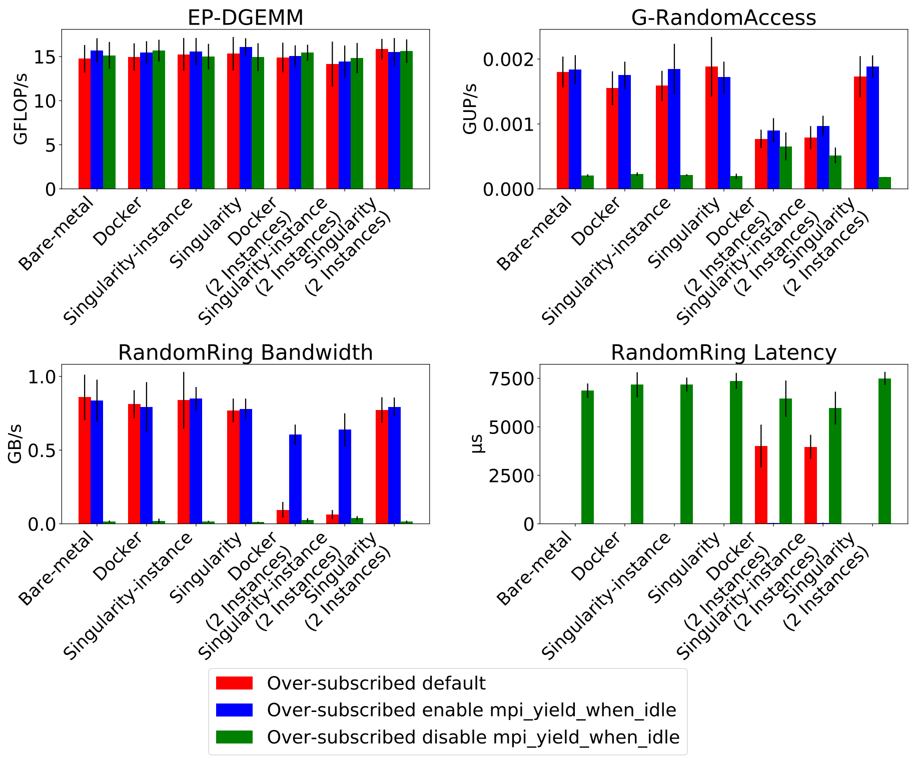


Figure 3.29: Comparison of over-subscribed mode on single and multiple container environments with different *mpi_yield_when_idle* configurations.

However, when using containers to run an MPI application in over-subscribed mode, things are more complicated. The difference between the aggressive and degraded modes in the OpenMPI engine when running on containers can be observed in Figure 3.29. For bare-metal, Singularity, and single-container deployments of Docker and Singularity-instance, the performance of the 'default' configuration matches with the performance when `mpi_yield_when_idle` is enabled, as the OpenMPI engine can automatically detect the over-subscription and run in degraded mode. However, for scenarios with multiple containers of Docker and Singularity-instance, the degraded mode must be set explicitly by enabling `mpi_yield_when_idle` in the `mpirun` command in order to let the process yield the processor to its peers. Otherwise, MPI processes running in disparate containers are not aware of their peers and will not yield the processor, thus degrading the performance ('default' configuration is degraded as when `mpi_yield_when_idle` is disabled). The results also show that, the most time the MPI processes are blocked in the MPI library, the most noticeable the benefits of declaring degraded mode operation are, indicating that MPI communication workloads will be especially sensitive to this.

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

3.3.5 Summary

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

- For Docker and Singularity-instance, multi-container deployments incur some performance degradation for MPI communication workloads, because the processes running on separated containers are deployed on isolated network namespaces and have to use the TCP/IP network stack rather than shared-memory to communicate with one another. This could be avoided by enabling shared-memory among the distinct containers and making the MPI engine aware of that shared-memory area [67].
- Singularity has close to bare-metal performance because the containers share the network and IPC namespaces and can use shared-memory to communicate the processes.
- Multi-container deployments of MPI throughput workloads do not incur significant performance degradation regarding bare-metal when increasing the number of containers due to the low amount of interprocess communication. Finer-grained deployments show a performance improvement because they simplify the scheduling in a similar way to when processes are pinned explicitly, which encourages further study of the impact of affinity on performance (see next section 3.4).
- On over-subscribed mode, some performance degradation is due to the scheduling of cgroups by Linux CFS, which results in an imbalanced allocation of processes to processors, because CFS is not especially accurate tracking the load of scheduling entities when they are groups of processes (*i.e.,* cgroups).
- The advantage/disadvantage of using the UMA hardware platform setting is not directly related with any containerization technology or container granularity, but with the application and hardware setting characteristics, such as the cache usage or the memory bandwidth. In particular, applications with low memory bandwidth requirements and good data locality perform better in the UMA setting, while memory-intensive applications perform better in the NUMA setting.
- It is necessary to enable MCA parameter `mpi_yield_when_idle` for multi-container deployments on over-subscribed mode, especially with MPI communication workloads, because this enables MPI processes on different containers to run in degraded mode.

3.4 Performance Analysis of Multi-container Deployments With Processor and Memory Affinity

This section addresses the second question: What is the impact of processor and memory affinity on the performance of multi-container deployment schemes for HPC workloads?

3.4.1 Objective

As mentioned in the introduction, Ibrahim et al. [67] have shown that the performance of HPC workloads on multi-socket NUMA architectures degrades when virtual machines span several NUMA domains. They claimed that the degradation was caused by the two-level memory management inherent in virtualized systems combined with the lazy page reclamation policies implemented in modern kernels. Our results in the previous section showed that the performance of HPC workloads running on a single container does not suffer such degradation when spanning several NUMA domains, basically because containers use only one-level memory management (the same as bare-metal processes). Consequently, partitioning HPC workloads into multiple containers and containing each one in a single NUMA domain through affinity is not expected to show noticeable benefits from a memory translation perspective for most of the benchmarks analyzed in this paper.

Nevertheless, the impact of container granularity on multi-container deployments with affinity can be significant depending on the CPU and memory usage characteristics of each benchmark. For example, restricting the range of possible CPUs to be assigned to the containers can help applications that suffer many CPU-migrations and context-switches. Restricting the memory access of the containers to the NUMA node where their CPUs belong can help applications that present an elevated number of remote memory accesses.

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

3.4.2 Method

Most containerization technologies use by default the namespace capability of the control groups, but utilize the resource control capability only when the user explicitly provides the parameters [141]. For example, considering the experiments in Section 3.3, from the application perspective, the workload is partitioned into several containers. However, from the kernel perspective, all of them are still sharing the same resources in the system (and competing for them). Thereby, the kernel has to arbitrate this competition to access the system hardware or software resources and multiplex the containers to ensure that all of them receive a fair share.

The purpose of processor and memory affinity is to reduce the number of kernel-level cycles spent due to the process preemption (*i.e.,* avoid CPU-migrations and context-switches) and due to the system calls (*i.e.,* exploit locality in data accessing). The

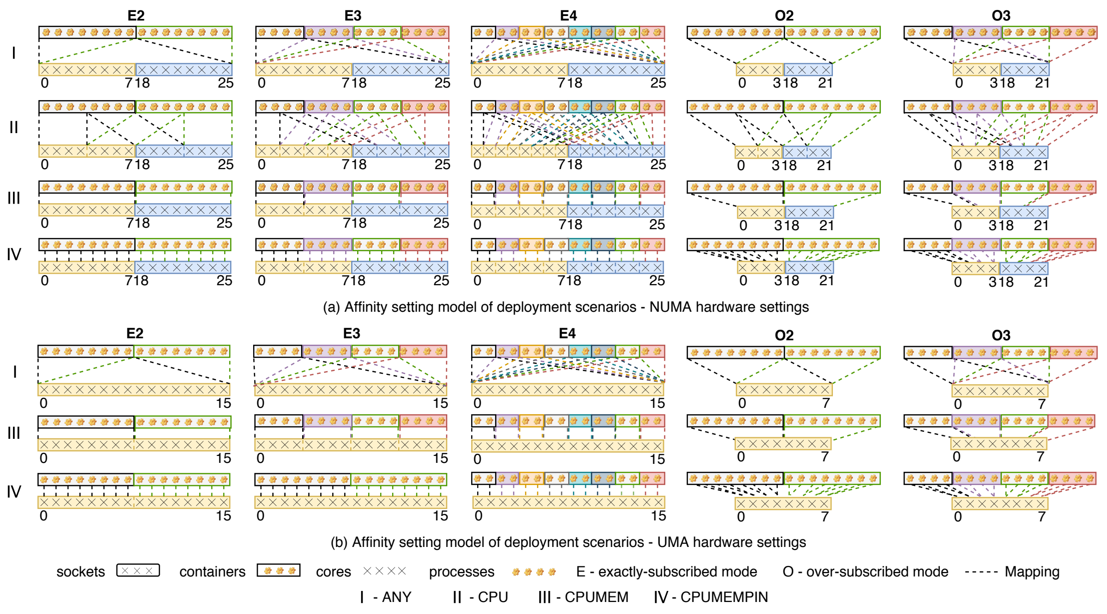

E2
E3
E4
O2
O3
I
II
III
IV
0
718
25
318
21
(a) Affinity setting model of deployment scenarios - NUMA hardware settings
15
7
(b) Affinity setting model of deployment scenarios - UMA hardware settings
sockets
containers
cores
processes
E - exactly-subscribed mode
O - over-subscribed mode
Mapping
I - ANY
II - CPU
III - CPUMEM
IV - CPUMEMPIN

Figure 3.30: Containerized deployment scenarios using affinity.

affinity settings for our containerized deployment scenarios are shown in Figure 3.30. They include three different settings, namely CPU, CPUMEM, and CPUMEMPIN, which are all compared to ANY (the baseline used in the experiments in Section 3.3). We assume a number of containers N_{ctn}, where each one hosts a number of processes N_{mpi}, so that $N_{ctn} \times N_{mpi} = K$, which is kept constant in all the deployment scenarios (*i.e.*, 16). For different subscription modes with ratio r, each container requests the number of cores $N_{cpu} = N_{mpi}/r$, where $r = 1$ or $r > 1$, which means the application runs on exactly-subscribed mode or over-subscribed mode, respectively. Each hardware platform setting provides a number of CPU cores and MEM nodes from one or more sockets $S = \{socket_s | s = 0, ..., N_{socket} - 1\}$, where each socket has P cores. Hence, for each application distributed in a set of containers $CTN = \{ctn_i | i = 1, ..., N_{ctn}\}$ where each one hosts a set of processes $MPI = \{mpi_j | j = 1, ..., N_{mpi}\}$, each affinity setting defines a mapping :

$$Map_{i,j} \rightarrow \begin{cases} CPU_{s,[x,y]} \\ MEM_s \end{cases} \tag{3.1}$$

where s refers to the assigned socket and $[x, y] = \{n \in \mathbb{Z} | x \leq n \leq y\}$ denote the assigned set of cores. Each of the affinity settings works as follows:

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

$$Map_{i,j} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{s,[s \times P, s \times P + \frac{N_{cpu} \times N_{ctn}}{N_{socket}} - 1]} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_s \end{cases} \tag{3.2}$$

(II) CPU: we define a specific processor affinity for each container to a set of cores from the two sockets available in the host. This can only be set in the NUMA hardware platform setting. The mapping of CPU scenarios could be formulated as follows:

$$Map_{i,j} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{s,[x_i,y_i]} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_s \end{cases} \tag{3.3}$$

$$x_i = s \times P + (i-1) \times \frac{N_{cpu}}{N_{socket}} \tag{3.4}$$

$$y_i = s \times P + i \times \frac{N_{cpu}}{N_{socket}} - 1 \tag{3.5}$$

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

$$Map_{i,j} \rightarrow \begin{cases} CPU_{s_i,[x_i,y_i]} \\ MEM_{s_i} \end{cases} \tag{3.6}$$

$$s_i = \lceil \frac{i}{N_{cps}} \rceil - 1 \tag{3.7}$$

$$x_i = s_i \times P + N_{cpu} \times ((i-1) - s_i \times N_{cps}) \tag{3.8}$$

$$y_i = s_i \times P + N_{cpu} \times (i - s_i \times N_{cps}) - 1 \tag{3.9}$$

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

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

$$Map_{i,j} \rightarrow \begin{cases} CPU_{s_i,[x_{i,j},y_{i,j}]} \\ MEM_{s_i} \end{cases} \tag{3.10}$$

$$s_i = \lceil \frac{i}{N_{cps}} \rceil - 1 \tag{3.11}$$

$$x_{i,j} = y_{i,j} = s_i \times P + N_{cpu} \times ((i-1) - s_i \times N_{cps}) + \lceil \frac{j}{r} \rceil - 1 \tag{3.12}$$

3.4.3 Experimental Setup

The environment, benchmarks, performance tools, statistical significance assessment methods, and container granularity settings are the same as Section 3.3.3. Some other settings regarding affinity are described below:

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

Memory affinity settings: The purpose of using memory affinity is to restrict the memory accesses of containers to the NUMA node where their assigned CPUs belong.

For Docker, the containers must be provided with the corresponding `cpuset-mems` parameter together with the `cpuset-cpus` parameter. For Singularity, we use the same strategy as Docker and specify `cpus` and `mems` options within the cgroups.toml file.

OpenMPI processes binding: Unlike the settings of ANY, CPU, and CPUMEM, where processes are free to be moved between the various CPUs allocated to each container, CPUMEMPIN utilizes bind-to core where a more rigid procedure of ranking, mapping, and binding of processes on CPUs is carried out, actually making it a 1-to-1 process-to-processor binding. For Docker and Singularity-instance, it was necessary to configure the appropriate rankfiles that describe this behavior.

3.4.4 Results

This section shows the impact when utilizing processor and memory affinity strategies on multi-container deployments of the HPCC benchmarks, using different containerization technologies and with two hardware platform settings(*i.e.,* UMA and NUMA settings described before).

① · Impact of containerization technology on multi-container deployments with affinity

Figure 3.31-3.35 show the performance results of MPI communication workloads. Congruently with the results in the previous section, Singularity achieves the best performance also when using processor and memory affinity, while Docker and Singularity-instance present some performance degradation in multi-container scenarios. As discussed in the previous section, this is due to the overhead of communication through the network stack instead of using shared-memory, which depends on the amount of time spent within the MPI library and the specific MPI functions invoked. Setting affinity cannot avoid this performance degradation.

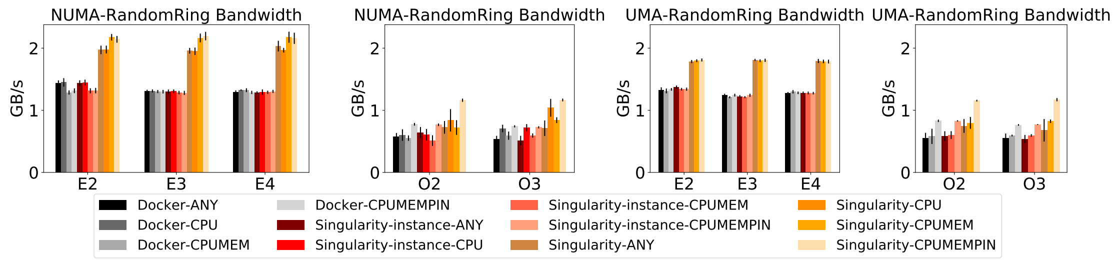


Figure 3.31: Impact of affinity on RandomRing-bandwidth performance.

Figure 3.36-3.37 depict the performance results of MPI throughput benchmarks. In this case, all the containerization technologies (Docker, Singularity-instance, and Singularity) achieve the same performance if they are set with the same affinity configuration. The effectiveness of using affinity with those benchmarks is not dependent on the containerization technology because, as we discussed in the previous section, they present low inter-process communication.

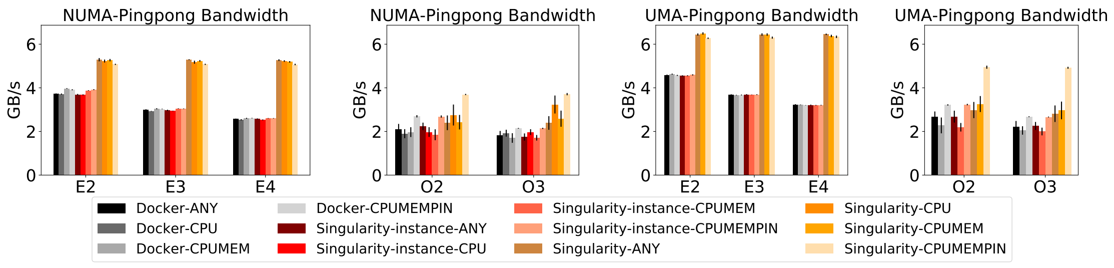


Figure 3.32: Impact of affinity on Pingpong-bandwidth performance.

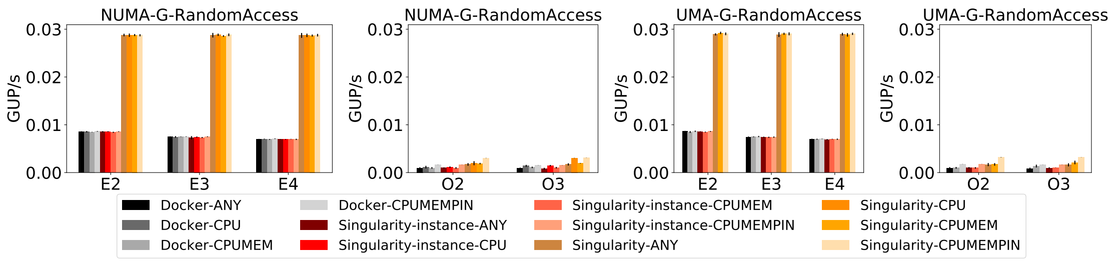


Figure 3.33: Impact of affinity on G-RandomAccess performance.

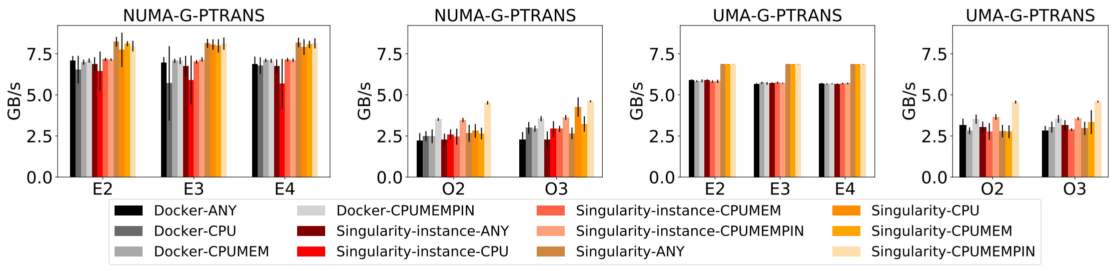


Figure 3.34: Impact of affinity on G-PTRANS performance.

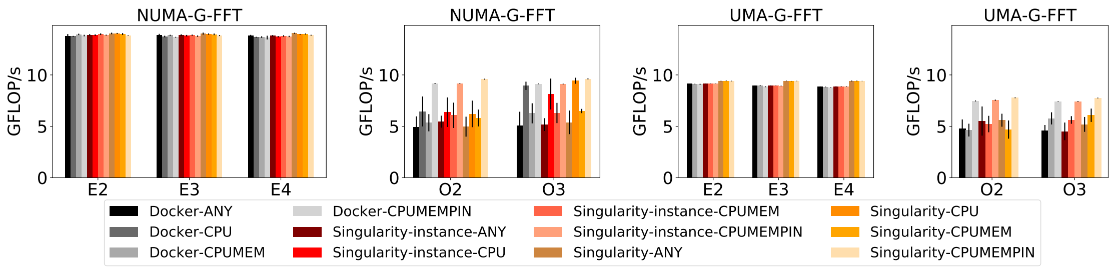


Figure 3.35: Impact of affinity on G-FFT performance.

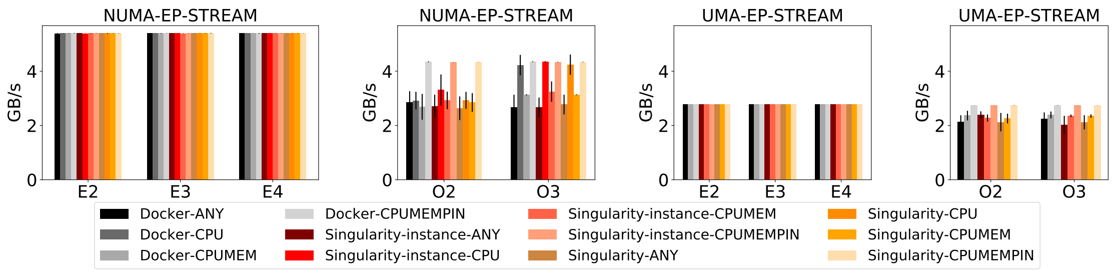


Figure 3.36: Impact of affinity on EP-STREAM performance.

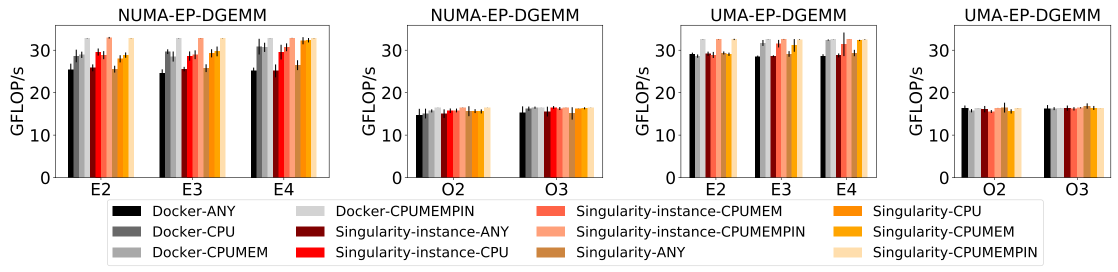


Figure 3.37: Impact of affinity on EP-DGEMM performance.

② · Impact of container granularity on multi-container deployments with affinity

As discussed previously, the impact of container granularity on multi-container deployments with affinity can be significant depending on the CPU and memory usage characteristics of each benchmark. The results in Table 3.6, which depict the access rate to the local memory in the NUMA setting for each benchmark on ANY scenario, show that EP-STREAM, G-PTRANS, G-FFT, and G-RandomAccess are well optimized for locality (processes mostly access the local memory), while EP-DGEMM has distributed memory allocation (only 56% accesses to local memory) (b_eff performs most of its accesses to remote memory, but as it uses few memory, this is not significant for performance). Consequently, only EP-DGEMM can take advantage of using memory affinity to reduce the latency to access the memory, and the benefit of memory affinity for the other benchmarks should be negligible. Similarly, as the local memory access rates are slightly lower on over-subscribed deployment scenarios than exactly-subscribed ones, over-subscribed mode scenarios have more room for exploiting better memory affinity.

Table 3.6: HPCC benchmark memory locality analysis.

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

CPU and memory affinity have considerably increased the performance of EP-DGEMM in all the scenarios. Specifically, the improvement (in %) for Docker in CPU, CPUMEM, and CPUMEMPIN scenarios with respect to ANY scenarios on the NUMA setting is significant in all the exactly-subscribed scenarios (with P-values of the corresponding T-tests ranging from $9.7e^{-11}$ to $1.8e^{-4}$, clearly below 0.05): around 12%–22% (E2–E4 CPU), 13%–21% (E2–E4 CPUMEM), and 29%–33% (E2–E4 CPUMEMPIN). In the over-subscribed mode, the improvement is also significant in O2-CPUMEMX and O3 scenarios (with P-values ranging from $7.6e^{-3}$ to 0.06): 7% (O2 CPUMEM), 11% (O2 CPUMEMPIN), 6% (O3 CPU), 7% (O3 CPUMEM), and 7% (O3 CPUMEMPIN), but not significant in O2-CPU: 2% with P-value 0.5. These performance increments are directly related with the container granularity, as finer-grained deployments provide better improvement. This happens because CPU affinity restricts the number of assigned CPUs within each container, hence the processes running in finer-grained containers have less available CPUs where they could be migrated. This can be seen in the counter values in Figure 3.38. Setting CPU affinity reduces the number of context-switches and CPU-migrations in CPUX scenarios, while setting memory affinity restricts as well the remote memory accesses in CPUMEMX scenarios. Overall, affinity improves the cache usage and optimizes the data allocation of the EP-DGEMM application.

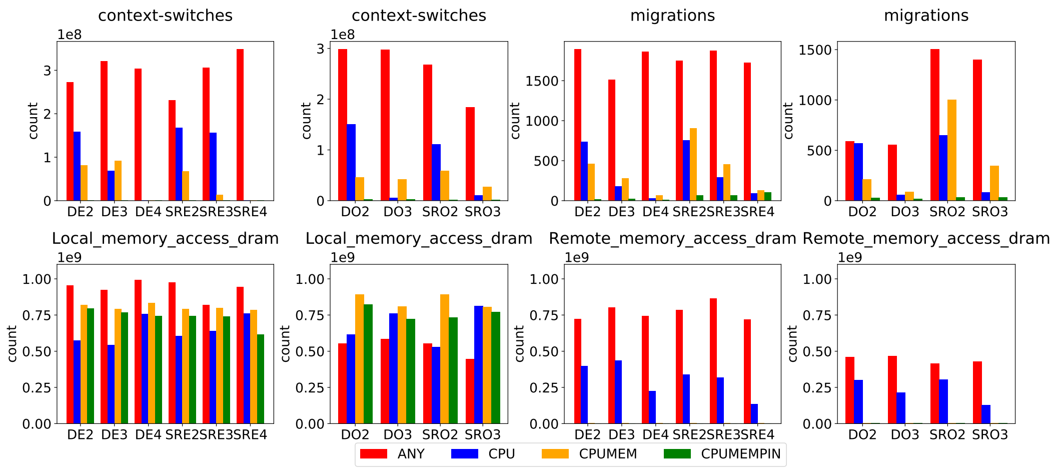


Figure 3.38: Performance event counters of EP-DGEMM on Docker and Singularity for scenarios with different affinity on NUMA hardware platform setting

For EP-STREAM, G-PTRANS, G-FFT, G-RandomAccess, and b_eff benchmarks, memory affinity does not impact significantly the performance because b_eff uses few memory and the others have the memory allocated mostly in the local socket already. The impact of CPU affinity on exactly-subscribed scenarios is not significant either for those benchmarks. As shown in Figure 3.39, which depicts the counter values for G-FFT benchmark on the NUMA setting, the operating system can do a pretty good job to prevent unnecessary context-switches on exactly-subscribed scenarios.

On the other side, CPU affinity can increase the performance in over-subscribed scenarios for those benchmarks. This is especially noticeable with CPUMEMPIN affinity configuration (*e.g.,* Docker shows significant improvements from 28% to 87% in scenario O2, with P-values clearly lower than 0.05 ranging from $6.7e^{-7}$ to $1.8e^{-4}$, and from 17% to 80% in scenario O3, with P-values also lower than 0.05 ranging from $8.3e^{-6}$ to $1.8e^{-4}$), and also with CPU configuration in scenario O3 (*e.g.,* improvements ranging from 31% to 77% are significant for all the benchmarks but b_eff(PingPong), with P-values ranging from $7.6e^{-4}$ to $1.9e^{-3}$). ANY configuration is also generally worse than CPU configuration in scenario O2 for all the benchmarks but b_eff(PingPong) (with improvements from 2% to 31%, but most of them not statistically significant as the P-values are higher than 0.05) and CPUMEM configuration in scenario O3(with improvements from 11% to 29%, which are halfway significant with P-values mostly ranging from 0.001 to 0.3). Results for CPUMEM in O2 are inconclusive, as all P-values of the T-tests are higher than 0.05, ranging from 0.2 to 0.79. As shown in Figure 3.39, in over-subscribed scenarios, CPUMEMPIN and CPU configurations have less CPU-migrations and context-switches than CPUMEM, which also has less than ANY. Processes in O3-CPU use cores belong-

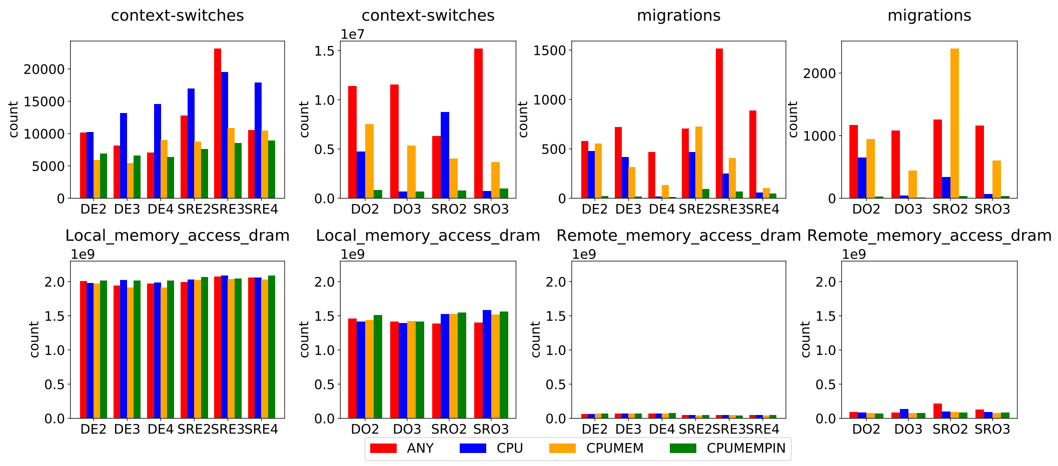


Figure 3.39: Performance event counters of G-FFT on Docker and Singularity for scenarios with different affinity on NUMA hardware platform setting

ing to two sockets. As migrations between sockets are more expensive (*e.g.,* expensive computation for iterating all the runqueues, expensive cache misses, and synchronization), the scheduler tries more to avoid them [105], something that does not happen in O3-CPUMEM, where the used cores belong to the same socket. Similarly, the improvement with CPU and CPUMEM in scenario O3 is also higher than in scenario O2, because O3 allows using only one core per socket, which is effectively encouraging 1-to-1 process-to-processor pinning.

③ · Impact of the cgroup scheduling on multi-container deployments with affinity

In section 3.3.4, we assessed the impact of the cgroup scheduling performed by CFS on ANY scenarios. CFS tried to maintain fair time allocation among cgroups, but incurred some performance degradation on over-subscribed mode scenarios due to load imbalance among the various processors. In this section, we assess the impact of the cgroup scheduling on multi-container deployments with affinity, to check if affinity could help to overcome this degradation.

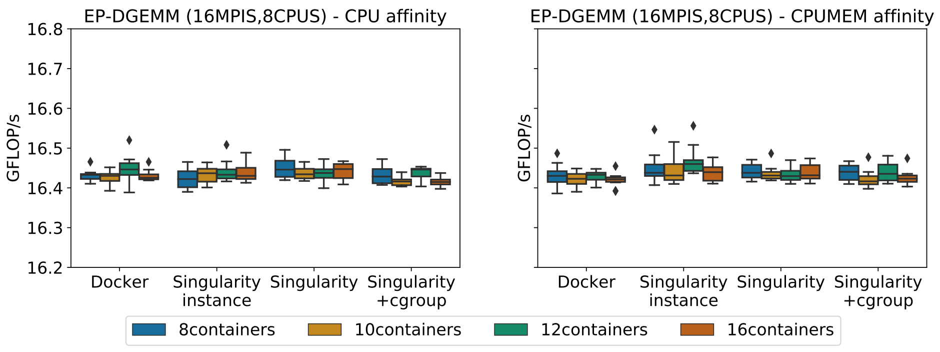


Figure 3.40: Performance comparison of EP-DGEMM on CPU/CPUMEM scenarios with different number of containers.

Figure 3.40 shows the EP-DGEMM performance on CPU and CPUMEM affinity scenarios with different number of containers. All the scenarios provide the same performance and do not incur performance degradation, even in containerization technologies which create a different cgroup per container (*e.g.,* Docker, Singularity-instance, Singularity+cgroup). This can be confirmed by means of T-tests for deployments with more than 8 containers regarding the 8-containers deployment, which have P-values ranging from 0.06 to 0.97, all higher than 0.05 and hence showing no significant difference. CPU affinity is able to overcome the CFS load imbalance problem because processes are deployed explicitly in fixed processors, which avoids load balancing by the scheduler.

④ · Impact of the hardware platform setting on multi-container deployments with affinity

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

3.4.5 Summary

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

- Multi-container deployments with affinity cannot prevent the performance degradation of Docker and Singularity-instance with MPI communication workloads due to the execution of containers on separated network namespaces. With MPI throughput workloads, all the containerization technologies achieve the same performance if they are set with the same affinity configuration.

- As containers do not virtualize memory, partitioning HPC workloads into multiple containers does not show benefits from a memory translation perspective, but finer-grained container granularity can improve the performance on multi-container deployments with affinity depending on the CPU and memory usage characteristics of each benchmark. Memory affinity reduces the number of accesses to the remote memory in benchmarks with distributed allocated memory, while CPU affinity restricts the cores that processes can be allocated, which reduces the number of CPU-migrations and context switches, especially in over-subscribed scenarios.

- 1-to-1 process-processor pinning scenarios (*i.e.,* CPUMEMPIN scenarios) provide the best performance, but less strict affinity configurations can be acceptable alternatives when 1-to-1 pinning is not straight-forward (*e.g.,* in over-subscribed scenarios where the number of processes is not a multiple of the number of processors).

- On over-subscribed mode, CPU affinity is able to overcome the CFS load imbalance problem causing performance degradation, because processes are deployed explicitly in fixed processors and this eliminates the need to balance load by the scheduler.

- Memory affinity does not provide added benefits in the UMA hardware platform setting, since memory accesses are already local. CPU affinity improves the performance of some benchmarks in over-subscribed scenarios (as in the NUMA setting), by reducing the number of CPU-migrations and context-switches.

3.5 Performance Analysis of Multi-container Deployments on InfiniBand Clusters

This section addresses the third question: What is the impact of container technologies, container granularity, processor and memory affinity on the performance of multi-container deployment scenarios using different network interconnects and protocols?

3.5.1 Objective

As shown in the introduction, a matter of the utmost importance for HPC users is that the containers running their applications can leverage the underlying HPC resources such as Infiniband networks, which offer high-speed networking capabilities with improved throughput and low latency through the use of Remote Direct Memory Access (RDMA) [204]. However, it is still unclear how multi-container deployment schemes with different affinity settings perform with various network interconnects and protocols, and how different communication patterns and message sizes impact the performance of containerized HPC workloads.

Based on the previous performance analyses, in this section, we present a detailed performance characteristic for HPC workloads on InfiniBand clusters. We consider different dimensions, namely network interconnects (including Ethernet and InfiniBand) and protocols (including TCP/IP and RDMA), networking modes (including host, MACVLAN, and overlay networking), different containerization technologies (including Docker and Singularity), container granularity and processor and memory affinity.

3.5.2 Method

Our performance characterization will consider the four dimensions in Figure 3.41, namely containerization technologies, networking modes, interconnects and protocols, and affinity, respectively.

Containerization Technologies: In this dimension, we choose Docker, Singularity, and its variant with container instances (hereinafter called Singularity-instance and

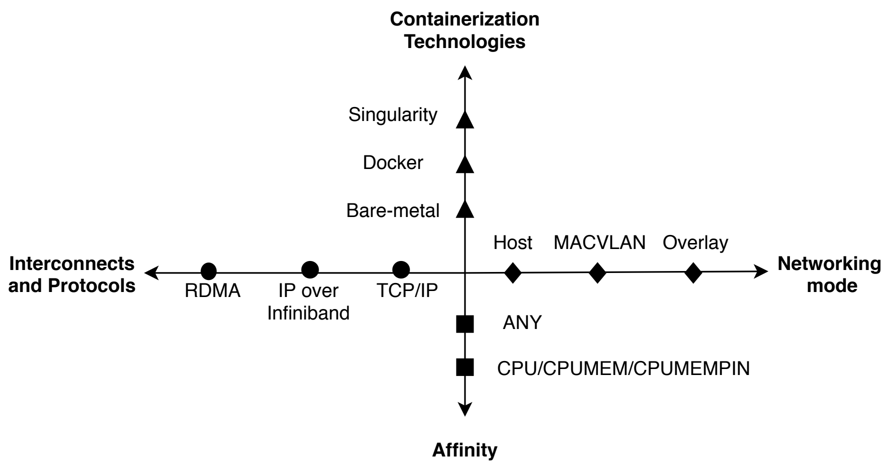


Figure 3.41: Four evaluation dimensions.

Singularity-instance+cgroup) as representative containerization technologies. The bare-metal performance is also provided to evaluate the corresponding overhead of each containerization technology.

Interconnects and Protocols: We consider 1-Gigabit Ethernet and InfiniBand interconnects in this dimension. We evaluate the performance of the different containerization technologies configured with several networking modes to operate on these interconnects through various protocols, such as TCP/IP and RDMA. Details are as we described in Section 2.1.1.

Networking mode: For Docker and Singularity-instance, the networking modes considered in the experiments depend on the number of deployed containers per host. As described in Section 2.2.3, when deploying a single container per host, we could use an underlay networking approach by sharing the host network with the containers or by setting a MACVLAN address to each container, or use an overlay networking approach through a network VXLAN tunnel that enables the communication across hosts. When deploying multiple containers per host, we can only use MACVLAN or overlay networking approaches for the communication of multiple containers across hosts.

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

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

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

$$Map_{h,i,j} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{h,s,[s\times P, s\times P+\frac{N_{cpu}\times N_{ctn}}{N_{socket}}-1]} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_{h,s} \end{cases} \tag{3.13}$$

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

$$Map_{h,i,j} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{h,s,[x_i,y_i]} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_{h,s} \end{cases} \tag{3.14}$$

$$x_i = s \times P + (i-1) \times \frac{N_{cpu}}{N_{socket}} \tag{3.15}$$

$$y_i = s \times P + i \times \frac{N_{cpu}}{N_{socket}} - 1 \tag{3.16}$$

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

$$Map_{h,i,j} \rightarrow \begin{cases} CPU_{h,s_i,[x_i,y_i]} \\ MEM_{h,s_i} \end{cases} \tag{3.17}$$

$$s_i = \lceil \frac{i}{N_{cps}} \rceil - 1 \tag{3.18}$$

$$x_i = s_i \times P + N_{cpu} \times ((i-1) - s_i \times N_{cps}) \tag{3.19}$$

$$y_i = s_i \times P + N_{cpu} \times (i - s_i \times N_{cps}) - 1 \tag{3.20}$$

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

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

$$Map_{h,i,j} \rightarrow \begin{cases} CPU_{h,s_i,[x_{i,j},y_{i,j}]} \\ MEM_{h,s_i} \end{cases} \tag{3.21}$$

$$s_i = \lceil \frac{i}{N_{cps}} \rceil - 1 \tag{3.22}$$

$$x_{i,j} = y_{i,j} = s_i \times P + N_{cpu} \times ((i-1) - s_i \times N_{cps}) + j - 1 \tag{3.23}$$

3.5.3 Experimental Setup

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

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

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

Benchmarks: *1) OSU Benchmark:* OSU Benchmark[5] is a suite of benchmarks that measure the MPI-level operation performance. We choose this benchmark for understanding MPI communication performance with different message sizes. We use version 5.6.3. *2) HPCC Benchmark:* The HPC Challenge benchmark suite[6] is widely used to evaluate the performance of HPC systems. Its design goal is to enable complete understandings of the performance characteristics of platforms [112]. It consists of several benchmarks that show the performance impact of real-world HPC applications. For example, the capability of processor floating point computation (*e.g.,* DGEMM, FFT), memory bandwidth (*e.g.,* STREAM, FFT) and latency (*e.g.,* RandomAccess), and communication bandwidth (*e.g.,* RandomRing Bandwidth, PTRANS, FFT) and latency (*e.g.,* RandomAccess) [197][62]. We use v1.5.0.

Networking mode and Protocol Settings: We evaluated various network interconnects and protocols, namely TCP/IP protocol on Ethernet, TCP/ IP protocol over InfiniBand (IPoIB), and RDMA natively on InfiniBand. Detailed network and protocol settings for each containerization technology are shown in Table 3.7. Single container per host scenarios are tested with three different networking modes: Host, MACVLAN, and Overlay. Note that MACVLAN does not work with InfiniBand, so we tested it only with TCP/IP on Ethernet. Multiple containers per host scenarios are tested only with the overlay networking mode, as this is the only mode that allows running multiple containers per host on all the network interconnects.

Docker implements its own networking specification called the Container Network Model[7], which supports multi-host networking through both underlay (based on MACVLAN) and overlay native drivers. The overlay network for Docker used in our experiments is not using Docker Swarm but configuring an external etcd[8] discovery service. For Singularity-instance, it uses the CNI[9] plugins for defining various basic networks such as bridge, ipvlan or macvlan. We use the knowledge from our previous work to enable the interconnection between Singularity instances across hosts [160]. As for Singularity-instance+cgroups, we keep the same network settings as Singularity-instance but enable the cgroup support by adding `apply-cgroups` parameter.

Granularity Deployment Scenarios: We study both single- and multi-container deployment schemes. One host acts as the master for launching the experiments and the other four hosts run each benchmark consisting of 128 processes in total. Detailed

[5]`https://mvapich.cse.ohio-state.edu/benchmarks/`
[6]`http://icl.cs.utk.edu/hpcc/`
[7]`https://github.com/docker/libnetwork/blob/master/docs/design.md`
[8]`https://etcd.io`
[9]`https://github.com/containernetworking/cni`

Table 3.7: Networking mode and protocol settings.

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

Table 3.8: Container granularity settings.

Containerization	# of Containers per Host (NC)	# of Processes per Container (NP)
Docker(D)	1,2,4,8,16,32	32,16,8,4,2,1
Singularity-instance(SI)	1,2,4,8,16,32	32,16,8,4,2,1
Singularity-instance + cgroups(SI+CG)	1,2,4,8,16,32	32,16,8,4,2,1

settings are shown in Table 3.8. For Docker, Singularity-instance, and Singularity-instance+cgroups, we generate scenarios SCE1–SCE6 by increasing the number of containers per host, in particular 1, 2, 4, 8, 16, and 32 containers per host, but decreasing the number of processes per container, that is, finer-grained container granularity (*i.e.*, 32, 16, 8, 4, 2, and 1 processes per container, respectively).

Scheduling and Binding Policy: OpenMPI's default mapping and binding policy schedules in a round-robin fashion through slots and automatically binds processes to sockets if the number of processes is more than two and binds processes to cores if the number of processes is less or equal than two. However, this binding policy is inadequate when enabling multi-container deployments because processes in different containers are not aware of their peers and always bind to the first socket by default. Thus, in experiments ① and ②, we use rankfiles with specific mappings between processes and cores to ensure a uniform distribution. For experiment ③, the rankfiles are derived from the formulas presented in Section 3.5.2. In addition, in our experiments, we restrict the resources to Docker and Singularity-instance+cgroups containers by setting `cpuset-cpus` and `cpuset-mems` parameters and specifying cpus and mems options within the cgroup file used by `apply-cgroups`, respectively.

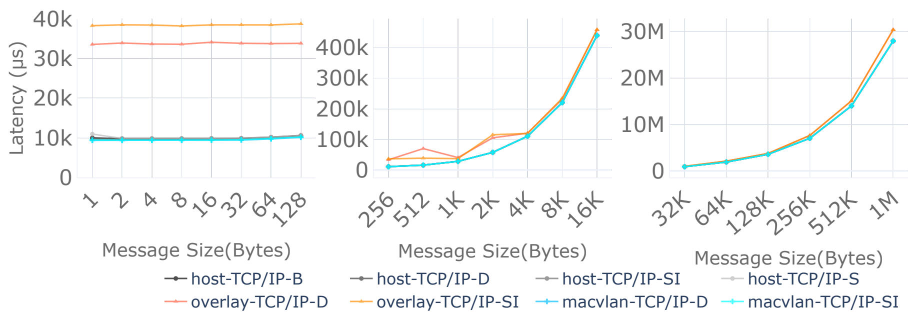
Performance analysis tools: We use Paraver[10] to profile MPI usage patterns of the benchmarks. We capture performance event counters and operating system metrics (through Perf[11]), such as context-switches, migrations, and memory accesses, from representative executions of the benchmarks and we use them to explain the obtained performance results.

[10] `https://tools.bsc.es/paraver`
[11] `http://man7.org/linux/man-pages/man1/perf.1.html`

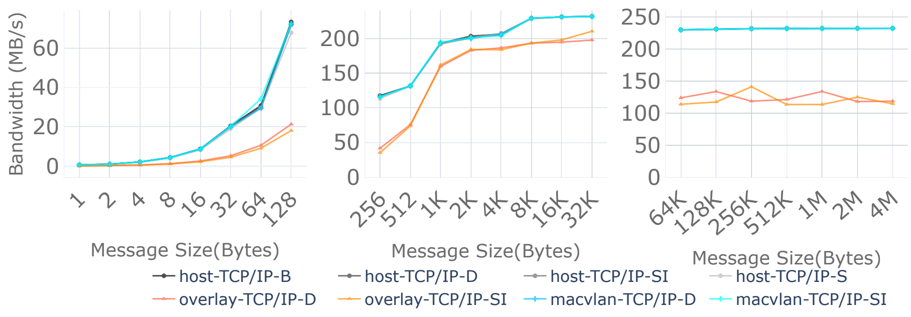
3.5.4 Results

① · Impact of Containerization on a Single Container per Host Deployment Scenario with Different Network Fabrics

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



(a) OSU MPI_Alltoallv Latency



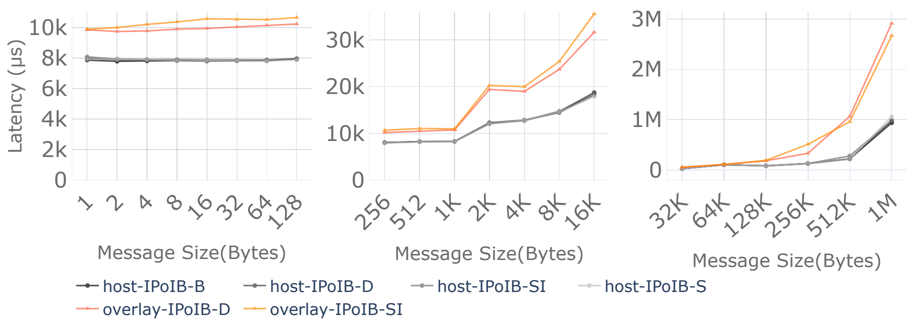
(b) OSU Bidirectional Bandwidth

Figure 3.42: **TCP/IP over Ethernet**: Latency (a) and Bandwidth (b) for scenario SCE1 with different networking modes.

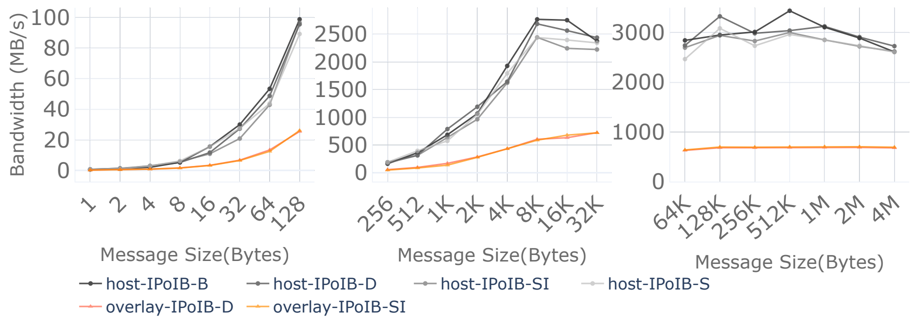
Figures 3.42–3.44 show the performance of different containerization technologies with several network fabrics and protocols. As expected, the RDMA protocol has higher performance and lower latency than IPoIB, and those two perform better than TCP/IP in all the container networking modes.

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

For Docker and Singularity-instance, underlay container networking approaches, such as host networking and MACVLAN networking, also achieve comparable performance to bare-metal experiments. With host networking, the single container shares the same

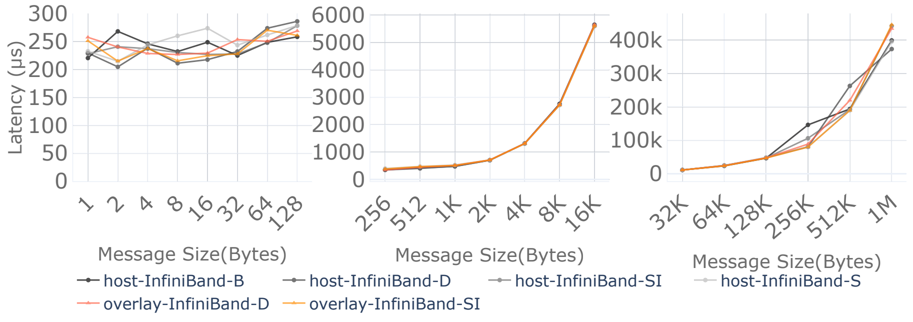


(a) OSU MPI_Alltoallv Latency

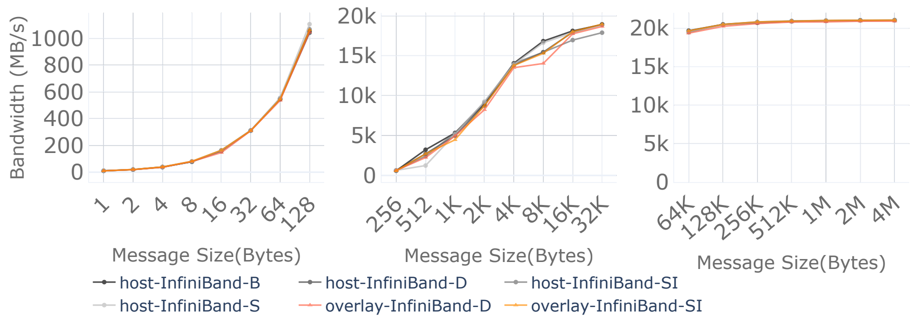


(b) OSU Bidirectional Bandwidth

Figure 3.43: **TCP/IP over Infiniband**: Latency (a) and Bandwidth (b) for scenario SCE1 with different networking modes.



(a) OSU MPI_Alltoallv Latency



(b) OSU Bidirectional Bandwidth

Figure 3.44: **RDMA over Infiniband**: Latency (a) and Bandwidth (b) for scenario SCE1 with different networking modes.

network namespace as the host. With MACVLAN networking, a container gets unique MAC and IP addresses and is exposed directly to the underlay network. In contrast, overlay networking brings explicit latency increase and bandwidth degradation for Docker and Singularity-instance. This occurs because all the communications among containers must be encapsulated through a tunnel, and this additional encapsulation incurs overhead (*i.e.,* reduces the amount of application data sent on each network packet). For TCP/IP over Ethernet, latency increments are more significant with small messages, whereas bandwidth degradation occurs for all message sizes. For example, Docker overlay networking shows 244% (8B), 9% (1MB) latency increase and 70% (8B), 49% (1MB) bandwidth degradation compared to bare-metal. For IPoIB, overlay networking shows significant latency increments (especially with large messages) and bandwidth degradation with all sizes compared to bare-metal. In particular, Docker presents 26% (8B), 211% (1MB) latency increase and 70% (8B), 74% (1MB) degradation on bandwidth. Both TCP/IP and IPoIB can benefit from an increment of the MTU (Maximum Transmission Unit) value to attenuate the incurred overhead by overlay networking for communication-intensive workloads.

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

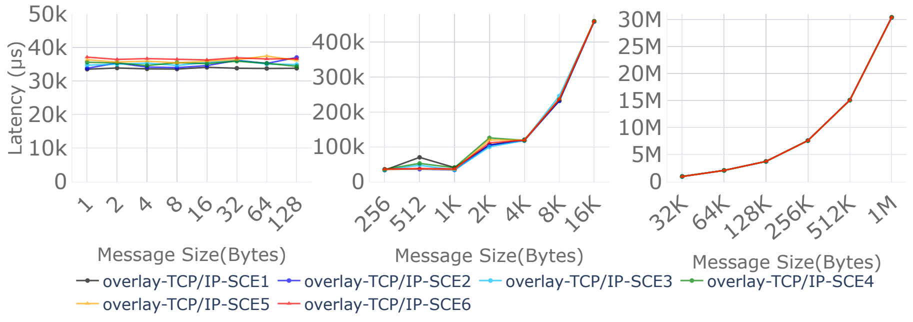
② · Impact of Container Granularity on Multi-container per Host Deployment Scenarios with Different Network Fabrics

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

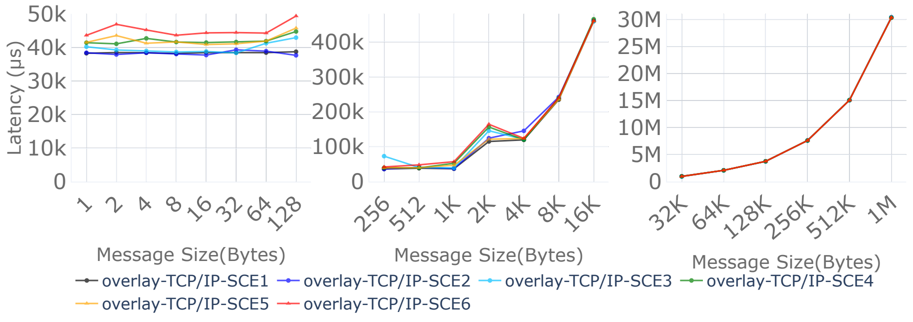
OSU MPI_Alltoallv Latency: Figures 3.45–3.47 show the MPI_Alltoallv latency of multi-container deployments for Docker and Singularity-instance with several network fabrics and protocols, namely TCP/IP, IPoIB, and RDMA.

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

For large messages, TCP/IP has similar performance with different container granularity. However, for IPoIB and RDMA, scenarios with several containers per host (SCE2–SCE6) show up to 19% and 10% lower latency than SCE1 for IPoIB and RDMA, respectively. This is because the memory latency becomes a critical factor when the network latency is not the dominant bottleneck, as occurs in high-speed networks. As shown in Figure 3.48, which depicts relevant performance counters of osu-alltoallv for IPoIB and

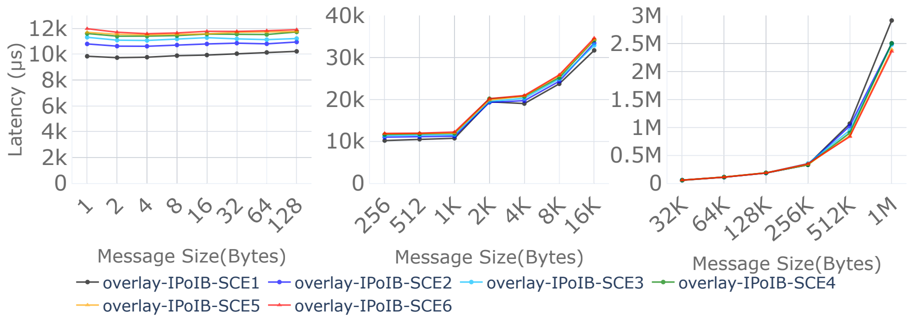


(a) MPI_Alltoallv latency on Docker

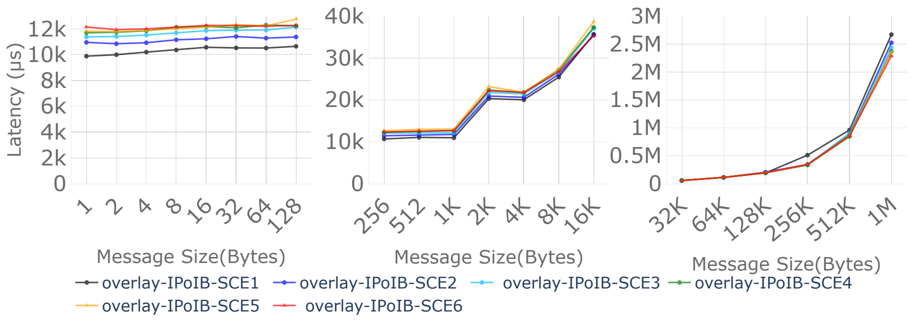


(b) MPI_Alltoallv latency on Singularity-instance

Figure 3.45: **TCP/IP over Ethernet**: MPI_Alltoallv latency for multi-container deployment scenarios (SCE1–SCE6).

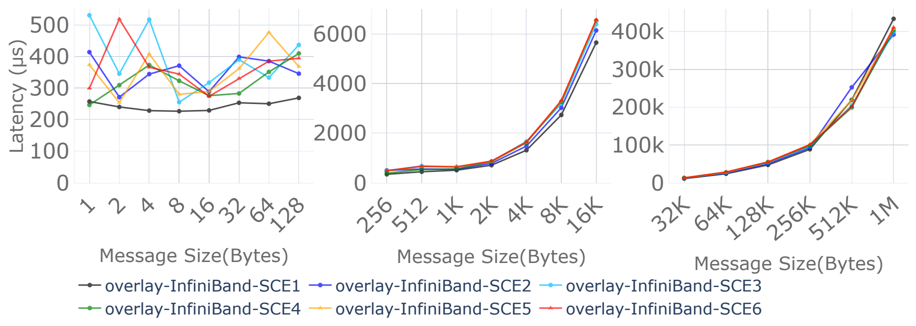


(a) MPI_Alltoallv latency on Docker

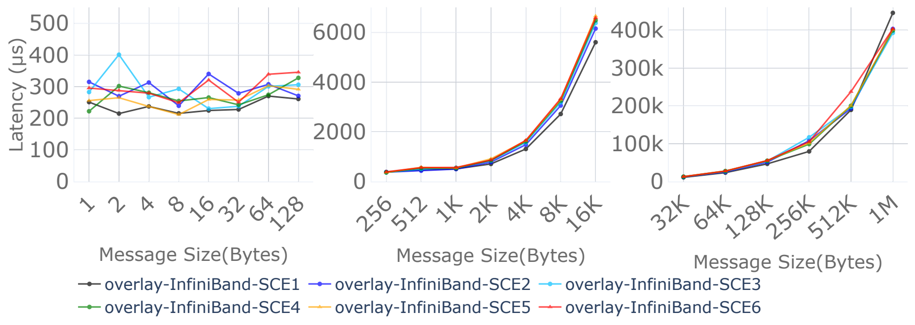


(b) MPI_Alltoallv latency on Singularity-instance

Figure 3.46: **TCP/IP over Infiniband**: MPI_Alltoallv latency for multi-container deployment scenarios (SCE1–SCE6).



(a) MPI_Alltoallv latency on Docker



(b) MPI_Alltoallv latency on Singularity-instance

Figure 3.47: **RDMA over Infiniband**: MPI_Alltoallv latency for multi-container deployment scenarios (SCE1–SCE6).

RDMA with large message size (1 MB) on Docker, scenarios SCE2–SCE6 show better cache utilization, fewer local memory accesses, and fewer remote memory accesses than SCE1. These are consequences of the scheduling of the containers (*i.e.*, the cgroups) and their corresponding MPI processes. With scenarios SCE2–SCE6 running more containers, each of them runs fewer processes, tending to a single-level scheduling (*i.e.*, at the cgroup level), which is simpler and allows exploiting processor affinity better, thus improving the cache usage and enforcing local memory accesses.

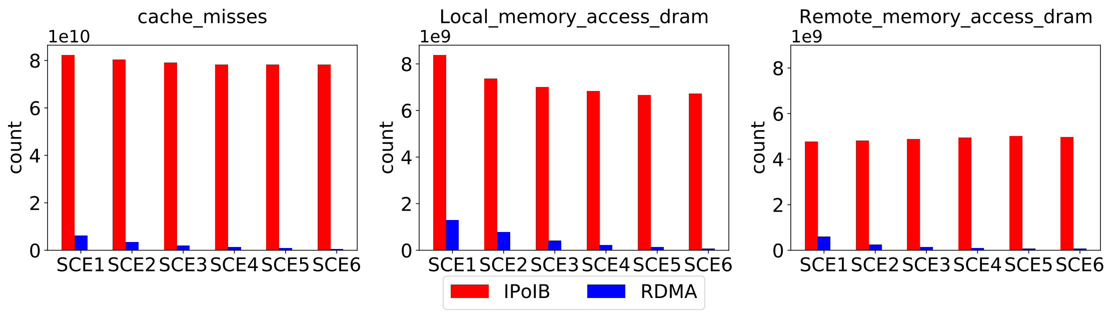


Figure 3.48: Performance event counters of osu-alltoallv for different interconnects and protocols with message size 1 MB on Docker.

The memory contention also affects the performance on IPoIB and RDMA. In order to measure the memory contention that occurs on osu-alltoallv with large message size, we calculate the memory contention ratio among cores by using the model proposed by

Tudor and Teo [182]. Like those authors, we are not interested in the absolute value of stall cycles, but on how stall cycles grow relative to a baseline value on one core (where there is no contention) due to memory contention among cores. Consequently, we derive the memory contention ratio ω as the stall cycles due to contention divided by the useful work cycles (including stall cycles that are not due to resource contention). Figure 3.49 presents the average memory contention ratio of osu-alltoallv for IPoIB and RDMA with message size 1 MB on Docker, where a higher ω means more memory contention. As shown in the figure, the memory contention ratio decreases when increasing the number of containers. This is because, as described previously, using more containers decreases the number of accesses to the l3 cache and the memory, which reduces the contention.

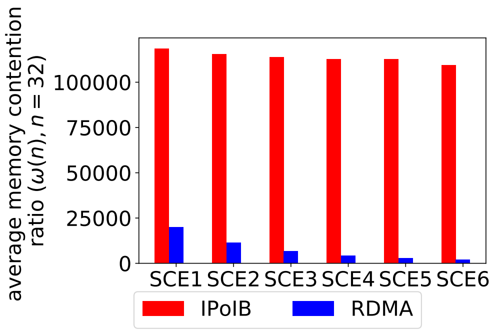
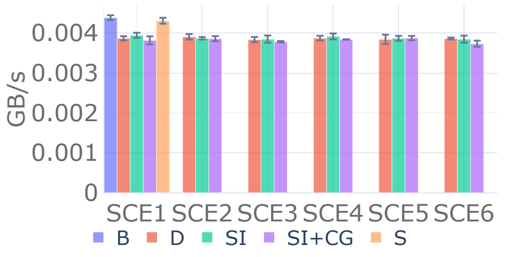


Figure 3.49: Average memory contention ratio of osu-alltoallv for different interconnects and protocols with message size 1 MB on Docker.

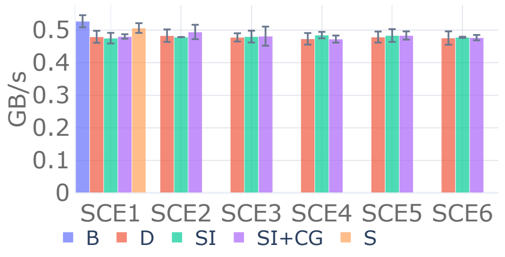
HPCC MPI communication-intensive workloads: RandomRing Bandwidth benchmark features a number of communication patterns (*e.g.,* non-blocking and blocking concurrent transfers). In particular, it performs MPI_Isend and MPI_Irecv to left and right partner, as well as MPI_Sendrecv, and saves the minimum of both latencies for all rings. We show the results for different container granularity in Figures 3.50(a), 3.51(a), and 3.52(a). Containerization technologies using overlay networks present significant degradation for TCP/IP and, especially, IPoIB regarding bare-metal and Singularity. As discussed in the previous section, this is due to the overhead introduced by the encapsulation of network packets.

For TCP/IP and IPoIB, increasing the number of containers per host does not have a noticeable impact on the bandwidth. This is because the bottleneck for TCP/IP and IPoIB comes from the interconnection between nodes, which is far slower than the interconnection between containers in the same node or between processes in the same container (*i.e.,* shared-memory). This can be confirmed in Figure 3.53(a-b). However, for RDMA, multi-container scenarios SCE2–SCE6 have 17%–23%–25%–26%–27% performance degradation in the bandwidth regarding SCE1. This occurs because the interconnection between nodes on RDMA is as fast as the shared-memory communication within a container as shown in Figure 3.53(c). Therefore, the interconnection between containers in the same node becomes the performance bottleneck, and this increases with the number of containers per node.

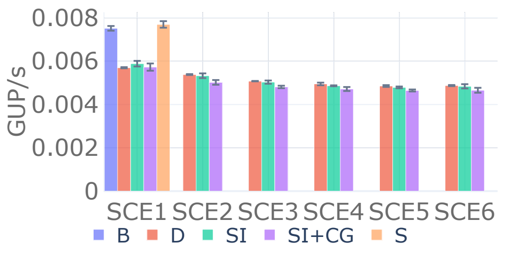
G-PTRANS and G-RandomAccess present different point-to-point communication patterns and use different message sizes. In particular, G-PTRANS performs mainly blocking concurrent transfers (*e.g.,* MPI_Sendrecv) with message size 2 MB. G-RandomAccess uses mostly small-sized non-blocking communication (*e.g.,* MPI_Isend, MPI_Irecv, MPI_Wait).



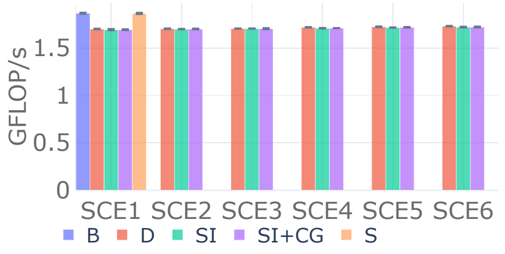
(a) RandomRing Bandwidth



(b) G-PTRANS

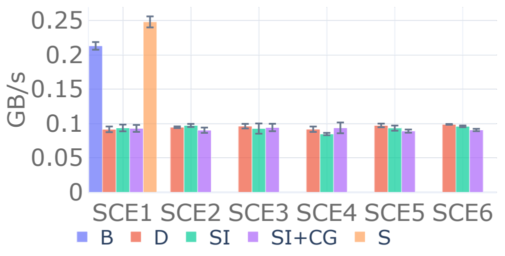


(c) G-RandomAccess

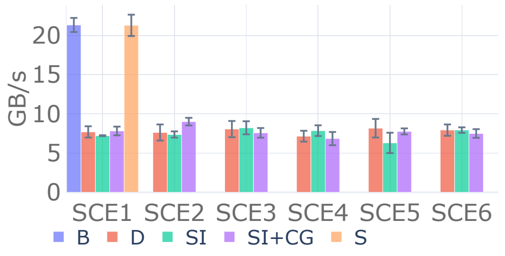


(d) G-FFT

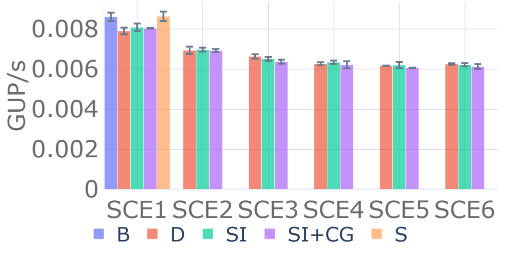
Figure 3.50: **TCP/IP over Ethernet**:Impact of container granularity in HPCC MPI communication workloads.



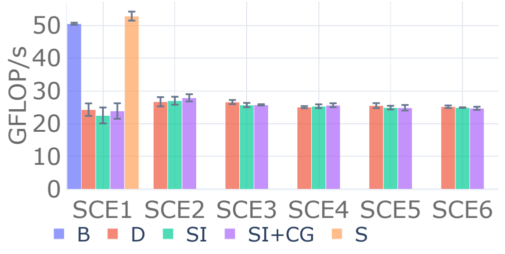
(a) RandomRing Bandwidth



(b) G-PTRANS

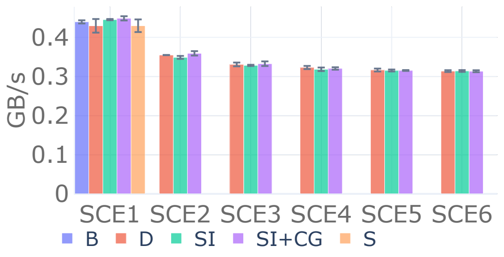


(c) G-RandomAccess

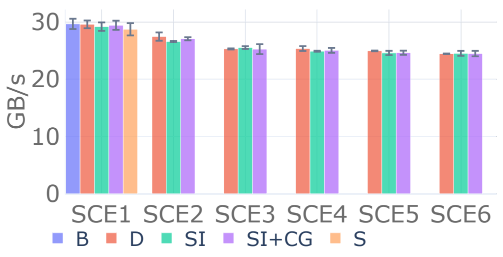


(d) G-FFT

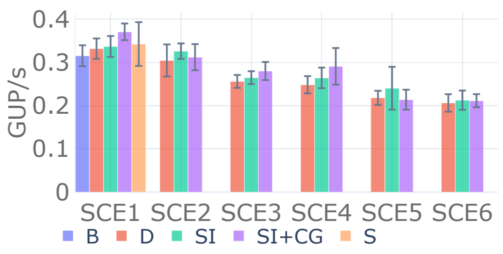
Figure 3.51: **TCP/IP over InfiniBand**: Impact of container granularity in HPCC MPI communication workloads.



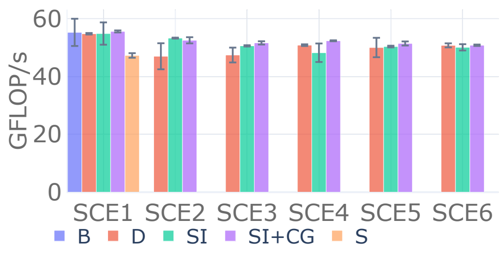
(a) RandomRing Bandwidth



(b) G-PTRANS

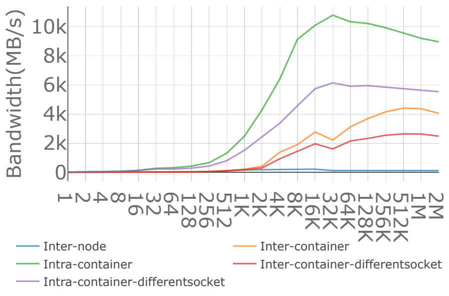


(c) G-RandomAccess

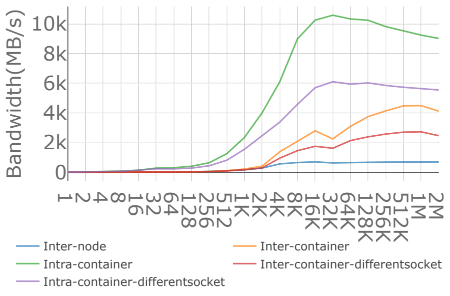


(d) G-FFT

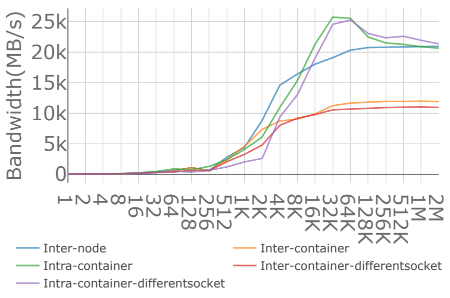
Figure 3.52: **RDMA over InfiniBand**: Impact of container granularity in HPCC MPI communication workloads.



(a) TCP/IP over Ethernet



(b) TCP/IP over InfiniBand



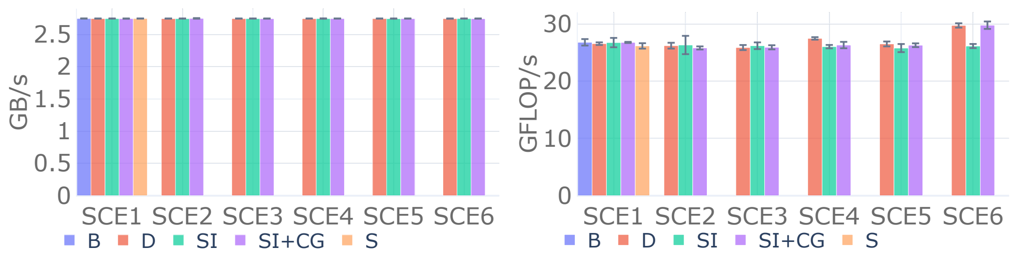
(c) RDMA over InfiniBand

Figure 3.53: Maximum aggregate bandwidth of inter-node, inter-container, and intra-container communications with different network protocols.

Thus, G-PTRANS is mainly a network-bandwidth-intensive benchmark that behaves similar to RandomRing. In particular, as shown in Figures 3.50(b), 3.51(b), and 3.52(b), Docker multi-container scenarios SCE2–SCE6 incur 7%–14%–14%–16%–17% performance degradation on RDMA compared to SCE1. On the other side, G-RandomAccess accesses data from all the processes. As shown in Figures 3.50(c), 3.51(c), and 3.52(c), there is an increasing performance degradation with finer-grained containers. In particular, Docker multi-container scenarios SCE2–SCE6 have 6%–11%–13%–15%–14%, 12%–16%–21%–22%–21%, and 8%–23%–25%–34%–38% performance degradation regarding SCE1, for TCP/IP, IPoIB, and RDMA, respectively. This occurs because G-RandomAccess performs a high number of MPI invocations, and when increasing the number of containers a significant part of them involve inter-container communications instead of intra-container (which are faster). This is especially relevant for RDMA, given that the memory latency is a critical parameter for the performance of G-RandomAccess.

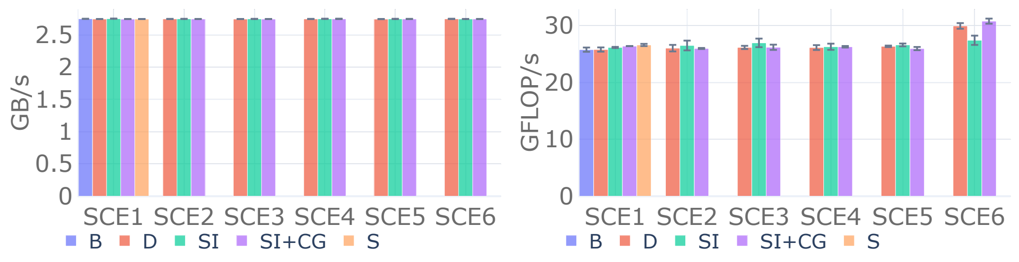
G-FFT mainly uses MPI_Alltoall communication pattern to transfer large data, and it is also intensive on memory bandwidth and computation. As shown in Figures 3.50(d), 3.51(d), and 3.52(d), the performance of multi-container scenarios SCE2–SCE6 is similar on TCP/IP and IPoIB, whereas on RDMA they show some performance degradation compared to SCE1 (*e.g.,* around 8% in average on Docker). Whereas the performance on TCP/IP (and IPoIB) is mostly limited by the network bandwidth, the performance on RDMA depends on the memory latency, which is worse when running multiple containers per host. However, this incurs low degradation due to the low number of MPI invocations performed by G-FFT.

HPCC MPI throughput workloads: EP-STREAM characterizes the memory bandwidth, while EP-DGEMM stresses the computation capabilities of the system. As shown in Figures 3.54–3.56, overlay networking does not bring explicit performance penalties due to the low amount of interprocess communication. Similarly, multi-container scenarios do not show significant performance differences, except EP-DGEMM on SCE6. In this scenario, EP-DGEMM on Docker (also on Singularity-instance+cgroup) shows noticeable performance improvement (11%, 16%, and 7% for TCP/IP, IPoIB, and RDMA, respectively) regarding other deployment scenarios (including bare-metal). This is a consequence of the scheduling of the containers (*i.e.,* cgroups) and their corresponding MPI processes. As each container runs a single process, this is essentially a single-level scheduling (*i.e.,* at the cgroup level), which is simpler and allows to exploit processor affinity better, in a similar way to when processes are pinned explicitly.



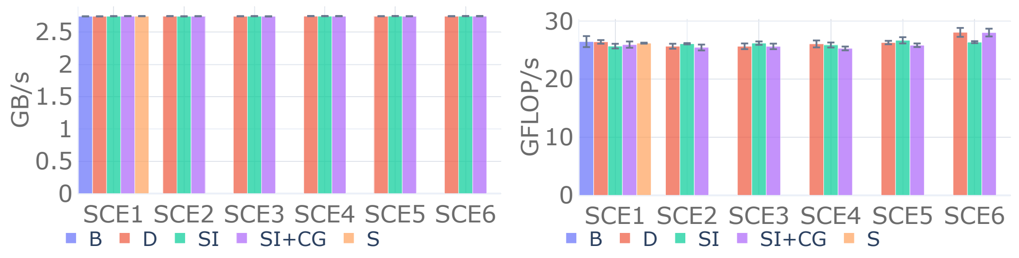
(a) EP-STREAM (b) EP-DGEMM

Figure 3.54: **TCP/IP over Ethernet**: Impact of container granularity in HPCC MPI throughput workloads.



(a) EP-STREAM (b) EP-DGEMM

Figure 3.55: **TCP/IP over InfiniBand**: Impact of container granularity in HPCC MPI throughput workloads.



(a) EP-STREAM (b) EP-DGEMM

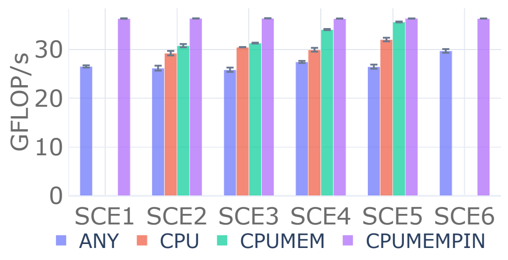
Figure 3.56: **RDMA over InfiniBand**: Impact of container granularity in HPCC throughput workloads.

③ · Impact of Affinity on Multi-container per Host Deployment Scenarios With Different Network Fabrics

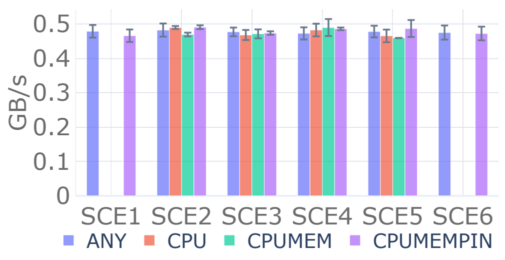
Figures 3.57, 3.58, and 3.59 show the performance results of Docker multi-container deployment scenarios with different affinity settings for various network interconnects and protocols. Singularity-instance shows similar results, which have not been included due to space constraints. As discussed in experiment ②, communication-intensive benchmarks have degradation in multi-container deployment scenarios due to the overhead of overlay communication for TCP/IP and IPoIB. Similarly, RDMA is limited by the bandwidth of inter-container communication. Setting affinity cannot avoid this performance degradation. For example, enabling affinity on G-PTRANS does not bring improvements because its performance is mainly limited by the network bandwidth.

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

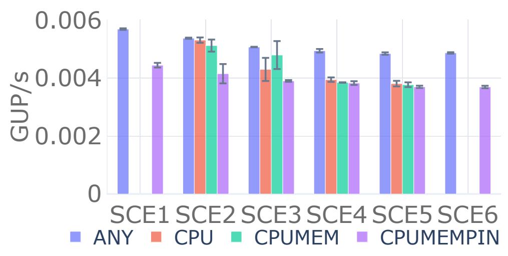
CPU and memory affinity have considerably increased the performance of EP-DGEMM in all the scenarios. Specifically, the speedup in CPU, CPUMEM, and CPUMEMPIN scenarios with respect to ANY scenarios ranges from 9%–21% (SCE2–SCE5 CPU), 18%–35% (SCE2–SCE5 CPUMEM), and 22%–41% (SCE1–SCE6 CPUMEMPIN) on TCP/IP; 15%–26% (SCE2–SCE5 CPU), 19%–36% (SCE2–SCE5 CPUMEM), and 21%–42% (SCE1–SCE6 CPUMEMPIN) on IPoIB; and 17%–22% (SCE2–SCE5 CPU), 21%–



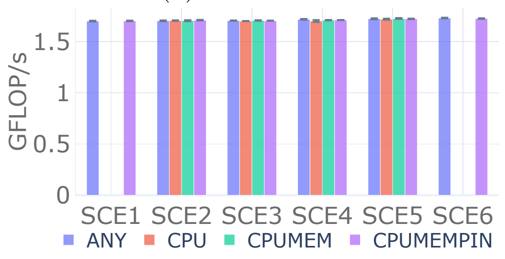
(a) EP-DGEMM



(b) G-PTRANS

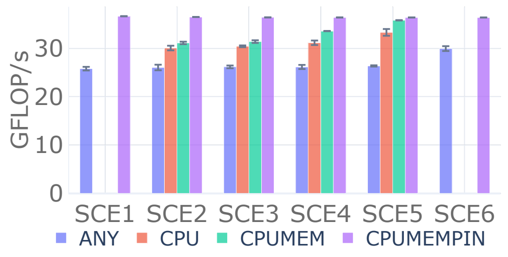


(c) G-RandomAccess

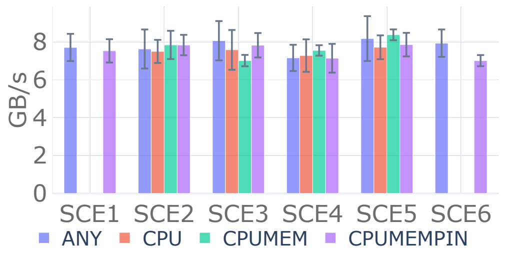


(d) G-FFT

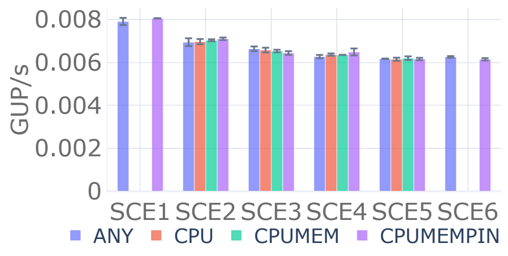
Figure 3.57: **TCP/IP over Ethernet**: Impact of affinity for multi-container deployments of HPCC MPI workloads.



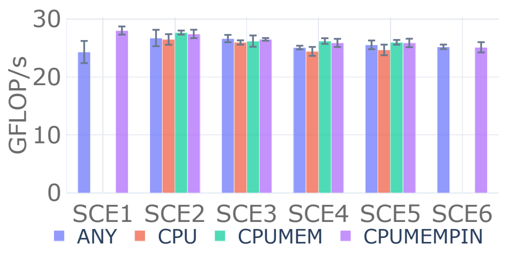
(a) EP-DGEMM



(b) G-PTRANS

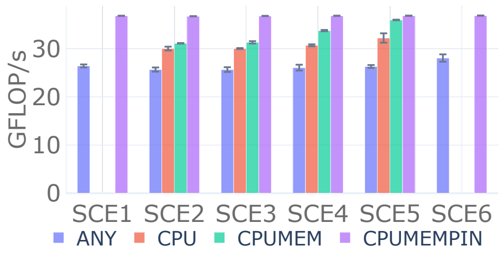


(c) G-RandomAccess

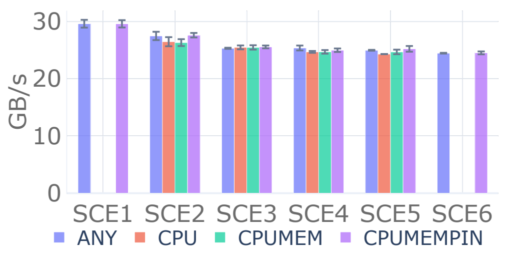


(d) G-FFT

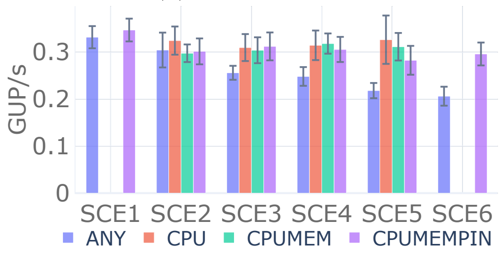
Figure 3.58: **TCP/IP over InfiniBand**: Impact of affinity for multi-container deployments of HPCC MPI workloads.



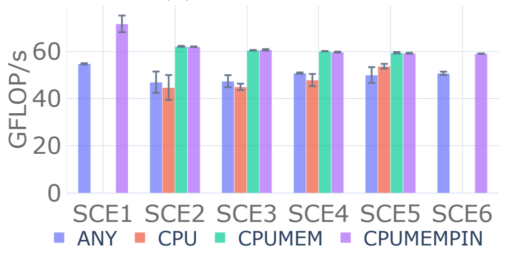
(a) EP-DGEMM



(b) G-PTRANS



(c) G-RandomAccess



(d) G-FFT

Figure 3.59: **RDMA over InfiniBand**: Impact of affinity for multi-container deployments of HPCC MPI workloads.

37% (SCE2–SCE5 CPUMEM), and 31%–43% (SCE1–SCE6 CPUMEMPIN) on RDMA. These performance increments are directly related with the container granularity, as finer-grained deployments provide better speedup on CPU and CPUMEM. This happens because CPU affinity restricts the number of assigned CPUs within each container, hence the processes running in finer-grained containers have less available CPUs where they could be migrated. Setting CPU affinity reduces the number of context-switches and CPU-migrations in CPUX scenarios, while setting memory affinity restricts as well the remote memory accesses in CPUMEMX scenarios.

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

Noticeably, setting affinity decreases the performance of G-RandomAccess on TCP/IP, up to 21% (CPU), 22% (CPUMEM), and 24% (CPUMEMPIN). By analyzing the results in Figure 3.60 and Figure 3.61, we found out that the actual cause of the performance degradation of CPUMEMPIN was the load imbalance among processes. Figure 3.60, which depicts the time spent in MPI communication patterns of G-RandomAccess for TCP/IP interconnect with ANY and CPUMEMPIN affinities on Docker scenario SCE1, shows that the time spent on MPI_Waitany and especially MPI_Barrier for CPUMEMPIN is much higher than ANY. Thus, this requires us to generate the MPI profile by duration time of the experiments.

Figure 3.61, which shows a detailed MPI duration profile for all 128 processes, reveals

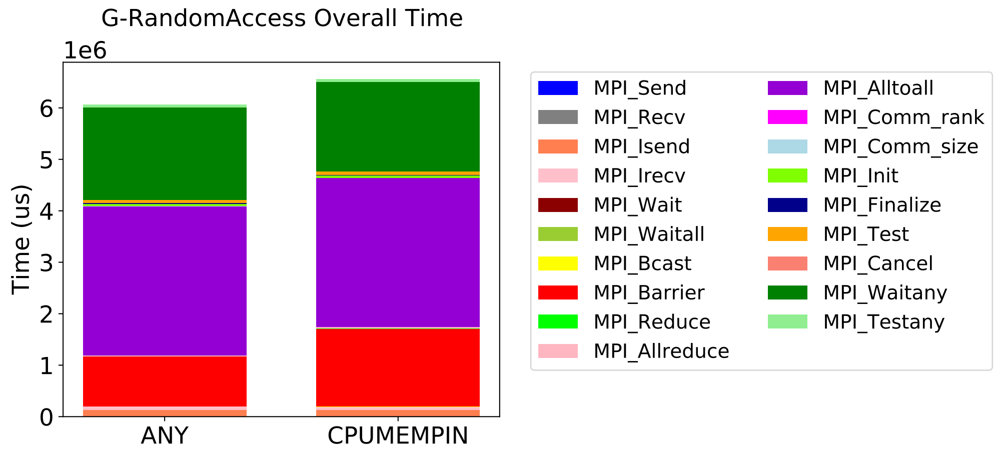


Figure 3.60: Time spent in MPI communication patterns of G-RandomAccess for TCP/IP interconnect with ANY and CPUMEMPIN affinity on Docker-SCE1.

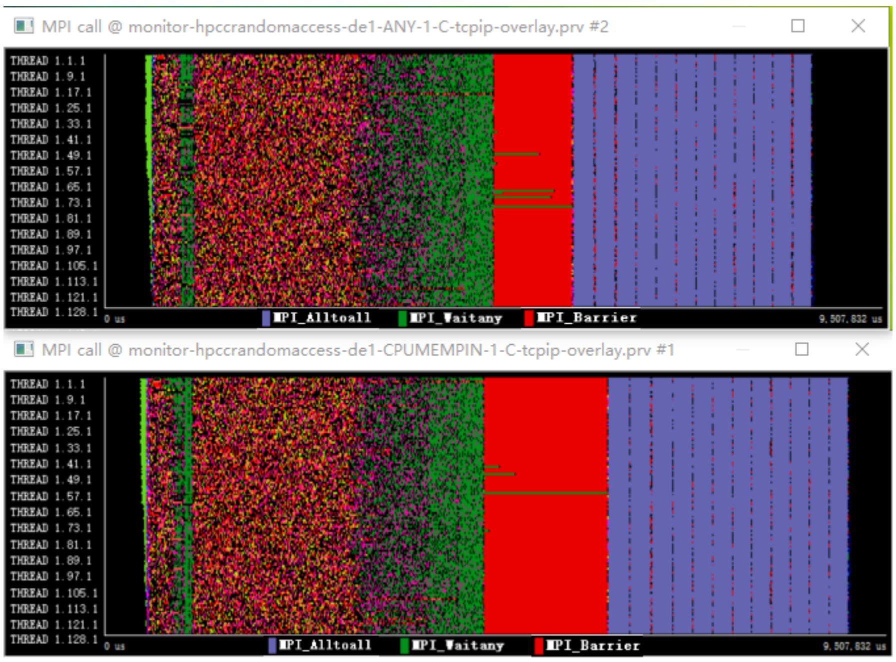


Figure 3.61: MPI profile of G-RandomAccess for TCP/IP interconnect with ANY (top) and CPUMEMPIN (bottom) affinity on Docker-SCE1.

that in ANY, the scheduler can better balance the load among processes and reduce their wait time in the barrier. Contrariwise, in CPUMEMPIN, some processes incur high wait latency that slows down the entire application. Given the random nature of the data accesses in G-RandomAccess benchmark, some processes might receive more requests than others, but as they are pinned to specific cores they cannot take advantage of other cores which are currently idle, thus causing the busy-waiting of other processes and introducing more latency. ANY affinity can mitigate this problem by allowing to migrate processes to achieve better load balance. However, enabling affinity can help to improve the performance on RDMA, in particular, 7%–50% (SCE2–SCE5 CPU), 0–43% (SCE2–SCE5 CPUMEM), and 0–43% (SCE1–SCE6 CPUMEMPIN). With RDMA, memory latency becomes relevant for performance, and for this reason, restricting the remote memory accesses through affinity can reduce the degradation.

④ · Performance Insights on Multi-container per Host Deployment Scenarios with Different Network Fabrics in Large-scale Clusters

Experiments in previous sections were run in a testbed with 5 nodes (1 master + 4 workers). Nevertheless, we anticipate that most of the performance insights obtained in those sections would still hold for multi-container deployment scenarios with different network fabrics in a large-scale cluster.

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

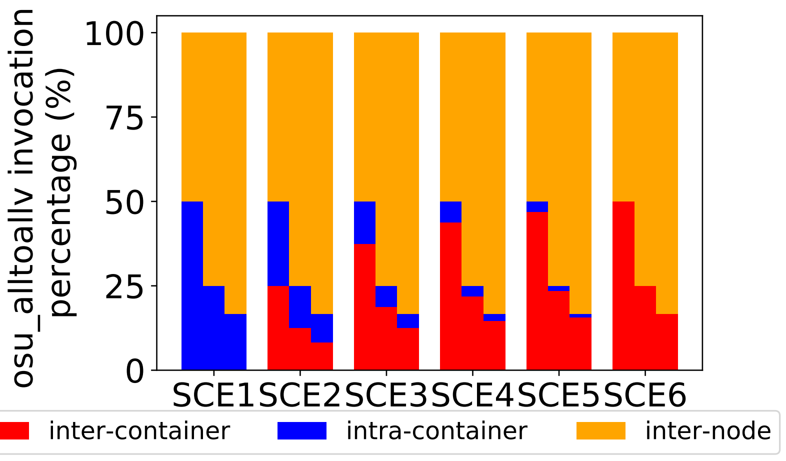
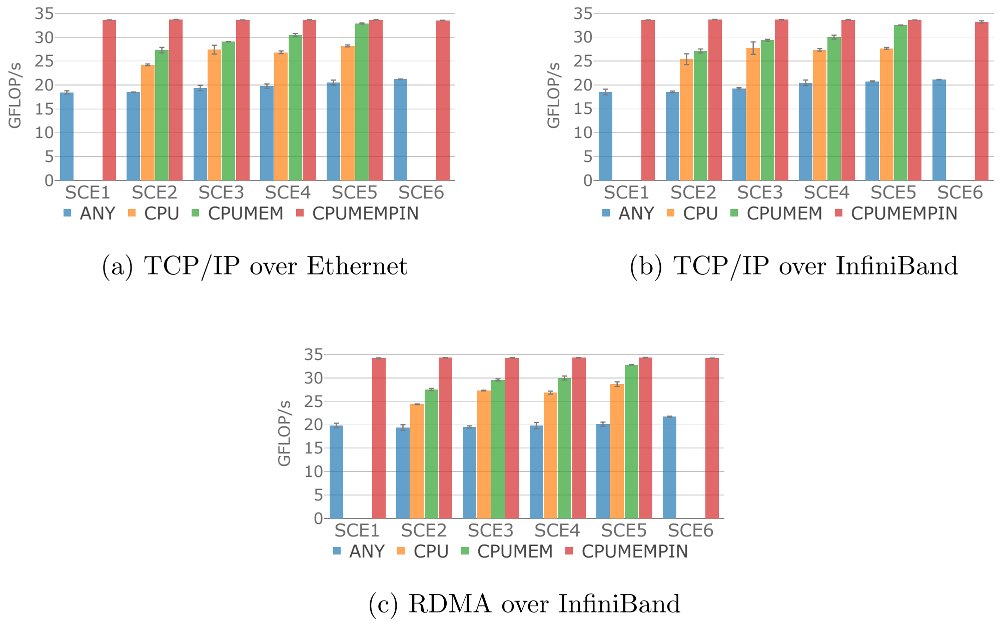


Figure 3.62: Distribution of invocations to alltoallv among inter-container, intra-container, and inter-node communications on a different number of nodes (each group shows the percentage for 2, 4, and 6 worker nodes, respectively).

Second, we expect Docker and Singularity-instance, which can support multi-container deployments by means of an overlay networking approach, to incur noticeable performance degradation for MPI communication workloads. This degradation is expected to increase as a function of the number of nodes, as this will increase the proportion of inter-node communications. The latter can be appreciated in Figure 3.62, which shows the distribution of invocations to function alltoallv in the OSU alltoallv benchmark among inter-container, intra-container, and inter-node communications with different number of nodes and deployment scenarios. Note that this benchmark communicates all the processes so the impact might vary depending on the communication pattern for other applications. It will basically depend on their ratio among inter-container, intra-container, and inter-node communications. Furthermore, the degradation will be more noticeable for TCP on Ethernet and on Infiniband, as they provide much worse performance than RDMA (check inter-node bandwidth in Figure 3.53) and the performance difference grows rapidly as the number of nodes increases [61].

As also shown in Figure 3.62, fine-grain multi-container deployments will transform intra-node communications using shared-memory on inter-container communications. Although Figure 3.53 showed that inter-container communications are slower, this effect will be diluted in large-scale clusters given the dominance of inter-node communications, hence multi-container deployments should show similar behavior in terms of the performance of deployment schemes with different container granularity.

Third, although setting affinity cannot avoid the overhead incurred by overlay networking, we expect that it can also make a difference on MPI throughput workloads in large-scale clusters, as the performance bottlenecks for those applications are the computation and memory allocation and not the network transfers. As shown in Figure 3.63, which displays the performance of EP-DGEMM using multi-container deployments scenarios when running on a testbed with 7 nodes (1 master + 6 workers), affinity still brings valuable performance benefits in all multi-container deployment scenarios and network fabrics.



(a) TCP/IP over Ethernet (b) TCP/IP over InfiniBand

(c) RDMA over InfiniBand

Figure 3.63: Performance of EP-DGEMM using multi-container deployments scenarios with different network fabrics when running on a testbed with 7 nodes (1 master + 6 workers).

3.5.5 Summary

The findings from the evaluation of the impact of container technologies, container granularity and processor and memory affinity on the performance of multi-container deployments using different network interconnects and protocols are as follows:

- Default Singularity reaches the same performance as bare-metal in all the scenarios, given that running on default Singularity is equivalent to running processes on bare-metal, as all the container processes on a given host reside in the same namespaces (*e.g.,* network, IPC, etc.) as the host. However, it does not support fine-grain multi-container deployments, which are only possible with Docker and Singularity-instance.

- For Docker and Singularity-instance, underlay container networking approaches, such as host networking and MACVLAN networking, also achieve comparable performance to bare-metal experiments. In contrast, overlay networking brings explicit latency increase and bandwidth degradation for Docker and Singularity-instance.

- Docker and Singularity-instance use an overlay networking approach, which incurs noticeable performance degradation for MPI communication workloads, and show similar behavior in terms of the performance of deployment schemes with different container granularity and affinity.

- Fine-grain multi-container deployments transform intra-node communications using shared memory on inter-container communications, which could increase the network latency of some MPI operations, but can alleviate the latency and contention of memory accesses when these are the performance bottleneck.

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

3.6 Conclusion and Future Work

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

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

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

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

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

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

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

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

In Chapter 6, we plan to use insights about the performance of multi-container deployments, especially those regarding the impact of the container granularity and the CPU and memory affinity, as well as the findings of the performance of multi-container deployments on InfiniBand clusters, especially those regarding the impact of the container granularity and affinity with different networking modes, to derive placement policies when deploying HPC workloads which can get better utilization of the resources while maintaining application performance. Those policies could be integrated into traditional HPC job schedulers, such as Slurm[12], which have also already started to support containers, as well as, new schedulers for HPC workloads with native containerization support, such as the Kubernetes native batch scheduling system (*i.e.,* Volcano[13]). Both approaches would allow integrating our deployment schemes, namely fine-grained container granularity, affinity, and overlay networking, with the traditional HPC scheduling capabilities and QoS requirements supported by those schedulers. Other directions, such as investigating larger-scale experiments beyond 6 workers or studying the performance of containerized HPC workloads over GPUs, are left as future work.

[12] `https://slurm.schedmd.com/containers.html`
[13] `https://volcano.sh/en/`

Chapter 4

Multi-Container Deployment Schemes for Online Machine Learning Inference

This chapter is based on the work done in collaboration with Amir Taherkordi during mobility for three months at the University of Oslo, the work has resulted in a publication in the IEEE CLOUD conference:

[**1**] Peini Liu, Jordi Guitart and Amir Taherkordi "Performance Characterization of Multi-container Deployment Schemes for Online Machine Learning Inference on Kubernetes Clusters", *2023 IEEE International Conference on Cloud Computing (CLOUD)*, July 2023, Chicago, USA. Accepted. (CORE RANK B).

This chapter presents multi-container deployment schemes for online ML inference on Kubernetes clusters. A brief introduction is presented in Section 4.1. Section 4.2 describes the architecture of our evaluated system and shows the detailed server and affinity setting schemes. Finally, the results of enabling multi-container and affinity deployments for ML workloads are shown in Section 4.3. The conclusions and the future work are described in Section 4.4.

4.1 Introduction

Machine Learning (ML) is increasingly becoming popular in various data analysis tasks such as image classification, machine translation, recommendation systems, and speech recognition [55][81][111][26]. *ML inference* is an important phase that uses trained ML models to make predictions from new data. From a runtime perspective, ML inference can be conducted either as a batch process, where predictions can be generated asynchronously from a batch of samples with no specific time limit to receive the results, or more interactively, through an online ML inference service, which receives dynamic queries from end-users and serves the predictions in real-time (subject to a latency bound) [97][98][203][92].

To meet the notable computational requirements of ML inference services, especially in the prediction step of the pipeline, those services are increasingly being deployed in the Cloud, which provides access to countless computational resources and allows to automatically scale the services by elastically deploying more or fewer instances to meet the changing demand. In this context, the objective of online ML inference service provisioning in the Cloud must be to find suitable deployment schemes such that inference services use the hardware efficiently and achieve the required performance (*e.g.,* throughput) to meet the dynamic queries from end-users.

To address this challenge, existing work considers online ML inference services provision and optimizations at different layers. In the application layer, different serving frameworks used by online ML inference services support configuration settings [144][176]. Experienced Data Engineers could tune the best parameter settings of these serving runtimes to improve the service performance [54]. In the infrastructure layer, the backends of a ML inference service can be horizontally- or vertically-scaled to use more resources[104]. These autoscaling frameworks [203][92][51] provide efficient ways for ML inference services to use resources while meeting Service Level Agreements (SLAs).

On top of that, current Cloud deployments are tightly coupled with containerization technology, which makes services easily reproducible and portable by encapsulating the code and dependencies. Furthermore, it isolates services so that they can be scaled or updated individually and failures do not affect the entire workload. Online ML inference services also aim to benefit from these features, to enable a seamless transition from training environments or to retrain (and redeploy) new models with the incoming new data, while meeting the performance requirements for the predictions. A typical application, in this context, is monitoring the performance of networks through analyzing the network traffic streams, which call for real-time and online learning data analytics and predictions [162].

However, there is limited knowledge about the impact of containerization on the performance of online ML inference services, and no well-defined guidelines on proper deployment schemes to exploit the potential of containerization and its capacity to constrain containers easily to a single NUMA (Non-Uniform Memory Access) domain or pin them to specific processors. In particular, *multi-container deployments* which partition the processes that belong to each application into multiple containers in each node are worth considering. Those deployments have been demonstrated to improve the performance of some multi-process HPC throughput workloads, which consist of the execution of loosely-coupled CPU-intensive processes in chapter 3 [99][101]. These characteristics resemble ML inference services, as numerous serving frameworks can exploit request-level parallelism to execute independent computationally-intensive prediction queries performed by various end-users through parallel threads.

In this chapter, we study suitable deployment schemes for allocating online ML inference services in the Cloud, focusing on container-level considerations (*i.e.,* fine-grained multi-container deployments and CPU/memory affinity settings). Our contributions are as follows:

- We define multiple deployment schemes for online ML inference services that feature different degrees of container granularity and we set the corresponding distribution of working threads and resources to each container to serve the model.

- We enable the definition of the CPU/memory affinity for each container belonging to an online ML inference service, as part of the former deployment schemes.

- We establish an evaluation system on a Kubernetes cluster and evaluate our multi-container deployments using typical ML inference benchmarks (*i.e.,* MLPerf) with different realistic client patterns.

- We present a systematic performance comparison, focusing on container-level considerations, to guide the Data Engineers on how to deploy their ML workloads to optimize the performance.

4.2 Evaluation Methodology

Our multi-container deployment schemes for containerized ML inference services are evaluated on a Kubernetes platform. This section describes the architecture of the evaluated system and the container granularity and affinity settings.

4.2.1 Evaluation System

The whole architecture of this system is depicted in Figure 4.1.

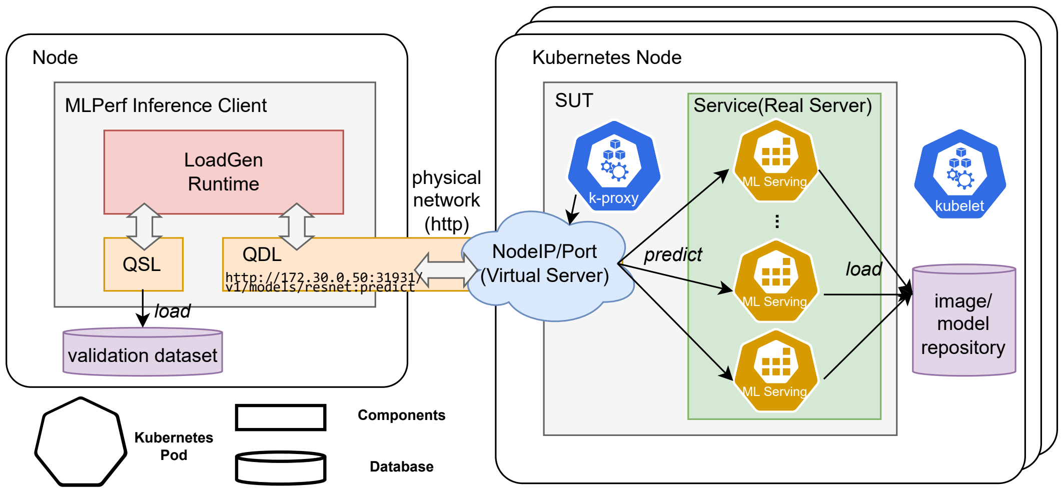


Figure 4.1: Evaluation system architecture of multi-container deployment schemes for ML model inference.

MLPerf Inference Client (LoadGen): MLPerf Inference is a benchmark suite for measuring how fast systems can run models in a variety of deployment scenarios [149]. LoadGen is the MLPerf client, which generates traffic for scenarios as formulated by a diverse set of experts, and efficiently and fairly measures the performance of ML inference systems. LoadGen is not dataset or model aware, so we need to define custom versions of the Query Sample Library (QSL) and the Query Dispatch Library (QDL) tailored to the datasets/models used in the paper. QSL is responsible for loading the data and includes untimed preprocessing. QDL is used to dispatch queries to the System Under Test (SUT) over a physical network, receive the responses, and pass them back to LoadGen.

System Under Test (SUT): The System Under Test refers to the ML inference system which provides an online ML inference service through several real server backends receiving queries from the client. In our experiments, SUT is established on a multi-core Kubernetes cluster, and models are served by Tensorflow Serving instances running inside multiple containers, each one wrapped as a Kubernetes Pod. Kubelet and Kube-proxy components from Kubernetes generate Pods on each node and distribute the queries among those Pods, respectively.

We consider various deployment options for the SUT depending mainly on two factors. First, the **container granularity** of the online ML inference service. In this paper, we assess the impact of deploying an online ML model inference service with different numbers of containers per host, that is, different multi-container deployment scenarios.

Second, the **resource affinity** of the containers running the online ML inference service. In this paper, we assess the impact of different CPU/memory affinity settings for each container.

4.2.2 Granularity Settings

Granularity settings define how we partition the online ML inference service into multiple containers (*i.e.* increasing the number of containers but decreasing the threads and resources on each container). A given SUT can have a single or multiple servers (each deployed within a container and with its own inference model), but the number of working threads and resources for the SUT are kept constant. We assume each SUT requires a number of CPU cores S_{cpu} and some amount of memory S_{mem} in GiB. Tensorflow Serving running within the SUT contains multiple working threads inside the server, namely inter-operation threads `tensorflow_inter_op_parallelism`, intra-operation threads `tensorflow_intra_op_parallelism`, and rest threads `rest_api_num_threads`. For each SUT, we define these numbers of threads as N_{inter}, N_{intra}, and N_{rest}.

(I) Multi-container deployments: Each SUT runs on a set of containers $CTN = \{ctn_i | i = 1, ..., N_{ctn}\}$ which use resources from a set of hosts $HOST = \{host_h | h = 1, ..., N_h\}$. Each container i has resources requirements $R^i_{\frac{S_{cpu}}{N_{ctn}}, \frac{S_{mem}}{N_{ctn}}}$ and a threading model $T^i_{\frac{N_{inter}}{N_{ctn}}, \frac{N_{intra}}{N_{ctn}}, \frac{N_{rest}}{N_{ctn}}}$, so that the total number of working threads and resources for the SUT are kept constant. Therefore, a multi-container deployment can be expressed as a set of containers each containing a subset of the threads and requiring a share of the resources.

$$SUT_{N_{cph}} = \bigcup_{i=1}^{N_{ctn}} ctn_i \rightarrow \begin{cases} R^i_{\frac{S_{cpu}}{N_{ctn}}, \frac{S_{mem}}{N_{ctn}}} \\ T^i_{\frac{N_{inter}}{N_{ctn}}, \frac{N_{intra}}{N_{ctn}}, \frac{N_{rest}}{N_{ctn}}} \end{cases} \tag{4.1}$$

where N_{cph} refers to the number of containers per host and is calculated as N_{ctn}/N_h.

(II) Baseline: This is the default strategy to deploy a SUT running Tensorflow Serving on Kubernetes. The baseline is deployed as a single-container-per-host deployment, thus it has $N_{cph} = \frac{N_{ctn}}{N_h} = 1$. The resources requirements for each container are calculated in the same way as with multi-container deployments. However, the threading model of Tensorflow Serving is decided by default: the threading pool size will be set to the number of visible cores within each server.

$$SUT_{baseline} = \bigcup_{i=1}^{N_{ctn}} ctn_i \rightarrow \begin{cases} R^i_{\frac{S_{cpu}}{N_{ctn}}, \frac{S_{mem}}{N_{ctn}}} \\ T^i_{default} \end{cases} \quad s.t. \ N_{cph} = 1 \tag{4.2}$$

4.2.3 Affinity Settings

Affinity settings define the exact resources from the hardware perspective that the containers of the online ML inference service will use. The affinity settings for our multi-container deployment scenarios are called *ANY* and *CPUMEM*. We assume a number of hosts N_h, and a number of containers N_{ctn}. The number of containers per host (*i.e.,* N_{cph}) is calculated as $N_{cph} = \frac{N_{ctn}}{N_h}$. The hardware platform provides a number of CPU cores and MEM nodes from one or several sockets $S = \{socket_s | s =$

$0, ..., N_{socket} - 1\}$, where each socket has P cores. Hence, for a set of containers $CTN = \{ctn_i | i = 1, ..., N_{ctn}\}$ which run on a set of hosts $HOST = \{host_h | h = 1, ..., N_h\}$, each affinity setting defines a mapping $Map_{h,i} \rightarrow CPU_{h,s,[x,y]} + MEM_{h,s}$ where h, s, and $[x, y] = \{n \in \mathbb{Z} | x \leq n \leq y\}$ denote the assigned host, socket, and set of cores, respectively. In particular, affinity settings ANY and $CPUMEM$ are defined as follows:

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

$$Map_{h,i} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{h,s,[s\times P, s\times P + \frac{N_{cpu}\times N_{cph}}{N_{socket}} - 1]} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_{h,s} \end{cases} \tag{4.3}$$

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

$$Map_{h,i} \rightarrow \begin{cases} CPU_{h,s_i,[x_i,y_i]} \\ MEM_{h,s_i} \end{cases} \tag{4.4}$$

$$s_i = \lceil \frac{i}{N_{cps}} \rceil - 1 \tag{4.5}$$

$$x_i = s_i \times P + N_{cpu} \times ((i - 1) - s_i \times N_{cps}) \tag{4.6}$$

$$y_i = s_i \times P + N_{cpu} \times (i - s_i \times N_{cps}) - 1 \tag{4.7}$$

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

4.3 Evaluation

In this section, we present an empirical performance evaluation of multi-container deployments of ML inference services on Kubernetes clusters. In this evaluation, we consider several schemes where we increase the number of containers serving the model but decrease the number of parallel working threads of the model per container. In addition, we consider different affinity settings and several real-world client scenarios.

4.3.1 Experimental Setup and Metrics

Hardware: Our experiments are executed on a five-node K8s cluster. Each host consists of 2 x Intel 2697v4 CPUs (18 cores each, hyperthreading disabled, CPU frequency scaling governor is set to max performance (*i.e.,* scaling_governor=performance)), 256 GB RAM, 60 TB GPFS file system, and 1-Gigabit Ethernet network.

Software: For all the hosts, we use CentOS release 7.7.1908 with host kernel 3.10.0-1062.el7.x86_64. The Kubernetes platform uses Kubernetes v1.19.16 (Docker v19.03.11, Etcd 3.4.9, Flannel 0.15.0, CNI 0.8.6, and CoreDNS 1.7.0). We use Tensorflow Serving v2.8.2 as the backend server and MLPerf Inference Client v0.7 (LoadGen) to emulate the clients.

Kubernetes Cluster Settings: Our Kubernetes cluster comprises five nodes. For each node, we reserve 4 cores for system and Kubernetes components, thus, 32 cores (16 from each socket) can be used for the allocation of ML inference services. By default, K8s Kubelet sets the CPU manager policy as 'none' which means all the containers can use the allocatable CPU resources within the resident node. For those experiments that require enabling CPU/memory affinity for containers, we configure Kubelet using `--cpu-manager-policy=static` and `--topology-manager-policy=best-effort`, which will start the containers on dedicated CPUs.

On the other hand, Kube-proxy is set to the IPVS (IP Virtual Server) mode which can direct requests for TCP- and UDP-based services to the real servers, and make services of the real servers appear as virtual services on a single IP address. The IPVS load balancing algorithm is kept as the default round-robin (rr) algorithm.

Tensorflow Serving Granularity Settings: Table 4.1 shows the different container granularity scenarios considered to deploy the online ML inference service, and the corresponding resources and thread pool size settings of each container.

$SUT_{baseline}$ is the baseline scenario which represents the basic deployment scheme of a Tensorflow Serving service. It normally contains one container per host and the container uses all the resources of the host. Each container also chooses its own thread settings, by default Tensorflow Serving will set the number of inter, intra, and rest threads as the number of visible cores within the container. In our case, even though the container can only use 32 CPUs (maximum available CPUs within one host), the threads will be set to 36 because the container can see all the cores in the host.

$SUT_{N_{cph}}$ refers to the various multi-container deployments of the Tensorflow serving service. For different granularity scenarios, we select a different number of containers to deploy the ML inference service, while partitioning the number of working threads and resources for each container. Thus, the total number of resources and threads for the inference service are kept constant in all the scenarios.

Affinity Settings: We consider two affinity settings: *ANY* and *CPUMEM*. The former means that all the containers can run on any CPUs and any memory node within hosts. The latter means that the containers will use dedicated CPUs and be bound to a specific memory node. These affinity settings are configured by an agent running on each node. For *ANY*, the agent will change all the containers' CPUSETs to a range of CPUs within a host, being the number of CPUs in the range equal to the number of requested CPUs per host. For *CPUMEM*, Kubelet is set to `--cpu-manager-policy=static` mode, thus each container is bound to dedicated CPUs (*i.e.,* different CPUSET) after its deployment. In addition, the agent will check the range of CPUs allocated to each container and set the corresponding memory node for this container.

MLPerf Inference Benchmark: As mentioned in section 4.2, our evaluation methodology uses the MLPerf Inference Benchmark, which is a benchmark suite specifically designed to measure the performance of ML models during inference. It includes standard models, datasets, and evaluation metrics of different client scenarios, which enables fair and comparable measurements.

i) Model and Dataset: MLPerf provides computer vision applications with its associated reference model (*i.e.,* a classifier network takes an image and selects the class that best describes it). In particular, for image classification, it provides a well-known vision model: the computationally-intensive Resnet50 [55] as a benchmark. This model accepts base64-encoded JPEG images as input and decodes them within the inference stage. We use the ImageNet 2012 dataset, crop the images to 224x224 in preprocessing,

Table 4.1: Server scenarios settings.

Scenarios ($SUT_{N_{cph}}$)	# of CTNs (N_{ctn})	Resources/CTN (R^i)	Threads/CTN (T^i)
$SUT_{baseline}$	$1 * N_h$	CPU=32cores MEM=128GiB	inter=36 intra=36 rest=36
SUT_1	$1 * N_h$	CPU=32cores MEM=128GiB	inter=32 intra=32 rest=64
SUT_2	$2 * N_h$	CPU=16cores MEM=64GiB	inter=16 intra=16 rest=32
SUT_4	$4 * N_h$	CPU=8cores MEM=32GiB	inter=8 intra=8 rest=16
SUT_8	$8 * N_h$	CPU=4cores MEM=16GiB	inter=4 intra=4 rest=8
SUT_{16}	$16 * N_h$	CPU=2cores MEM=8GiB	inter=2 intra=2 rest=4
SUT_{32}	$32 * N_h$	CPU=1core MEM=4GiB	inter=1 intra=1 rest=2

and send the strings of base64-encoded images through the physical network using REST APIs.

ii) Client Scenario Settings: MLPerf LoadGen provides four realistic end-user scenarios, namely Single-Stream (SS), Multi-Stream (MS), Server (S), and Offline (O), which represent many critical inference applications. The explanation of these client scenarios can be found in section 2.1.3. Additional details can be found in [149]. Table 4.2 summarizes their settings in our experiments, which we describe briefly below.

4.3.2 Multi-container Deployment and Affinity Evaluation on a Single Host

Figure 4.2 shows the impact of container granularity and affinity in SUT performance on different client scenarios on a Kubernetes cluster with a single node. The results are derived from 10 executions. Additionally, for some scenarios, we also analyze the inference time and the issue delay time for each individual sample.

SingleStream

This scenario generates low load because the client sends queries one by one, thus every time only one query is being processed at one of the containers of the SUT. From SUT_1 to SUT_{32}, that is, when deploying more containers per host (*i.e.,* from 1 to 32), each one has lower allocated resources (*i.e.,* from 32 CPUs/128 GiB to 1 CPU/4 GiB) and working threads (*i.e.,* from 32 to 1). *SingleStream* does not fully show the benefits of using multiple containers to deploy the online ML inference service, because always only one backend is used at a time, that is, we can only exploit parallelism within a request, not among requests.

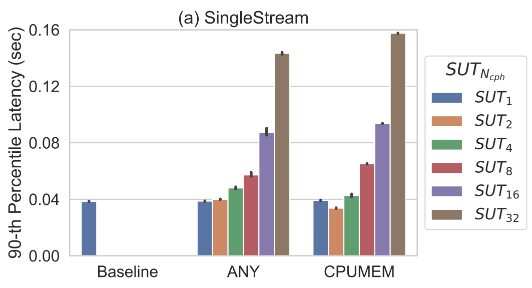
Table 4.2: Client scenarios settings.

Scenarios	**Query Generation**	**Metric**	**Sample per Query**	**Parameters**
Single-Stream (SS)	Sequential	90th-percentile Latency	1	min_query_count=1664

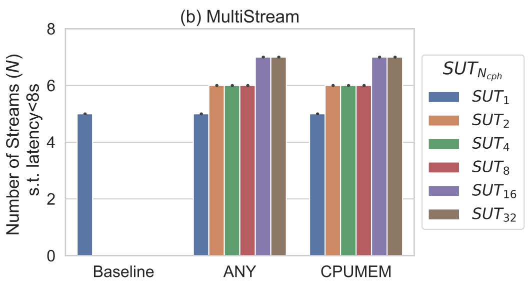
Multi-Stream (MS)	Arrival Interval With Dropping	Number of Streams Subject to Latency Bound	N	min_query_count=2000
				target_qps=32
				max_async_queries=256
				target_latency=8s

Server (S)	Poisson Distribution	Queries per Second Subject to Latency Bound	1	min_query_count=12800
				target_qps=200
				target_latency=20s

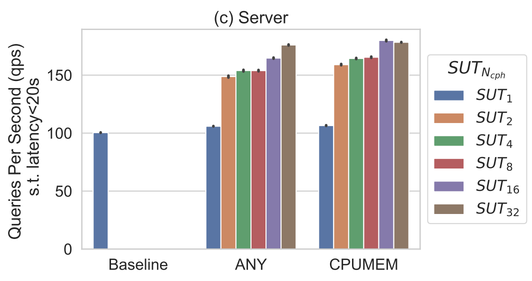
Offline (O)	Batch	Throughput	≥ 24576	min_query_count=32768
				target_qps=200
				max_batchsize=1,2,4,8



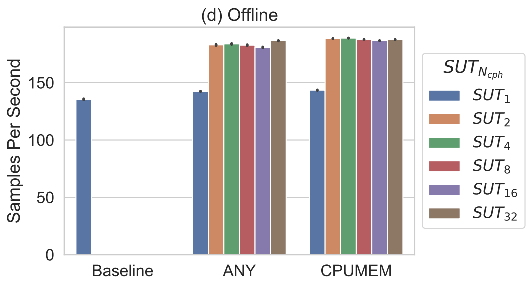
(a) SingleStream



(b) MultiStream



(c) Server



(d) Offline

Figure 4.2: Impact of container granularity and affinity in SUT performance on different client scenarios on a Kubernetes cluster.

Figure 4.2a shows the 90th percentile latency of different SUT deployments at *SingleStream*. For *ANY* scenario, running more containers per host increases the 90th latency (*i.e.,* SUT_2–SUT_{32} increase by 4%–25%–46%–122%–271% with respect to SUT_1). This increment is caused by the lower amount of resources and threads for each container as we increase the number of containers. On the other side, SUT_2 with *CPUMEM* settings shows 13% 90th latency improvement regarding SUT_1, because running two containers, one in each socket, improves the cache usage and avoids remote memory accesses between two NUMA nodes. However, SUT_4–SUT_{32} still show 7%–67%–140%–303% degradation regarding SUT_1 because the better locality cannot compensate for the lower parallelism as we increase the number of containers (due to the reduction of resources and threads per container).

Similarly, when comparing *CPUMEM* and *ANY* settings, the former shows better performance in coarse-grained scenarios $SUT_2 - SUT_4$ because each container has enough resources and threads to exploit the parallelism of the NUMA node to which they are assigned while getting the corresponding locality benefits. However, *CPUMEM* shows worse performance in finer-grained scenarios SUT_8–SUT_{32} because each container has less resources and threads but, still, *CPUMEM* allocates them in dedicated CPUs from two NUMA nodes. Contrariwise, *ANY* settings allow the containers to be allocated in the entire range of available CPUs, and due to the lower amount of resources each container needs, the scheduler is able to consolidate all of them in a single NUMA node.

MultiStream, Server, and Offline

The impact of container granularity and affinity in SUT performance on *MultiStream* (MS), *Server* (S), and *Offline* (O) client scenarios is shown in Figure 4.2b displays the maximum number of streams (subject to 99th latency < 8s) at *MultiStream*, Figure 4.2c displays the Queries per Second (qps) (subject to 99th latency < 20s) at *Server*, and Figure 4.2d displays the samples per second (the max_batch_size is set to 8 to optimize the performance in this scenario) at *Offline*. For comparison purposes, we also display detailed qps of *MultiStream* scenario in Figure 4.3. The three client scenarios show different patterns to send queries, but all of them generate a high load to the SUT, which consumes high computation resources, and allows to evaluate the impact of multi-container deployments.

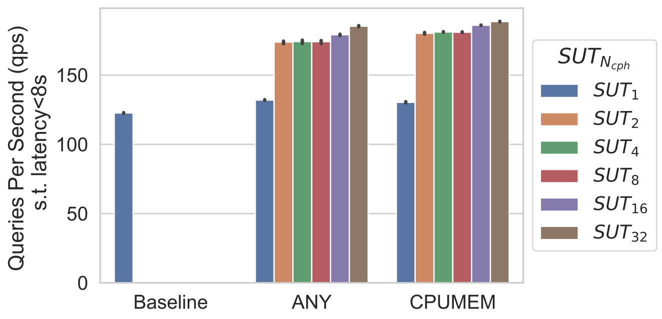


Figure 4.3: Queries Per Second in *MultiStream* ANY and CPUMEM scenarios.

Baseline: SUT_1 *ANY* and *CPUMEM* have roughly the same performance improvement up to 8%, 6%, and 6% compared to the baseline in client scenarios MS (see Figure 4.3), S (see Figure 4.2c), and O (see Figure 4.2d), respectively. This is because a single container in the baseline starts on all the CPUs within the host and Tensorflow Serving creates as many threads as visible CPUs (*i.e.,* 36) within this container, whereas there are effectively only 32 CPUs available for ML inference in the host (we reserve 4 cores for Kubernetes and system). Thus, the baseline has more CPU migrations and context switches among more threads than SUT_1 with *ANY* or *CPUMEM*, which start a single container on 32 cores and threads.

Granularity: Regarding the container granularity, in MS (see Figure 4.3), SUT_2–SUT_{32} have 32%–32%–32%–36%–40%, and 38%–39%–38%–42%–45% performance improvement regarding SUT_1 with *ANY* and *CPUMEM*, respectively; in S (see Figure 4.2c), SUT_2–SUT_{32} have 40%–45%–45%–56%–66% and 49%–55%–55%–69%–67% performance improvement regarding SUT_1 with *ANY* and *CPUMEM*, respectively; in O (see Figure 4.2d), SUT_2–SUT_{32} have 28%–29%–28%–27%–31%, 31%–32%–31%–30%–31% performance improvement regarding SUT_1 with *ANY* and *CPUMEM*, respectively. All the client scenarios show better performance with multi-container deployments. The difference is greater as we increase the number of containers for *MultiStream* and, especially, for *Server* scenarios. As shown in Figure 4.4, which displays the mean latency in *Server* scenarios, multi-container deployments show up to 90% latency improvement with respect to SUT_1, and finer-grained containers (from SUT_2 to SUT_{32}) show increasingly better performance.

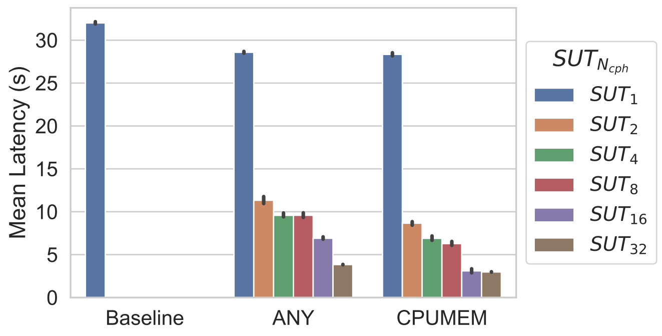


Figure 4.4: Mean latency in *Server* ANY and CPUMEM scenarios.

In *Server* scenarios, the overall latency of a sample can be broken down into the sample wait time before being processed and the actual sample inference time. As shown in Figure 4.5 and Figure 4.6, which display the inference time for individual samples in this scenario with *ANY* and *CPUMEM* settings, respectively, finer-grained multi-container deployments can use better the resources, and thus process the samples quicker, which reduces their inference time (and consequently, their latency). Note how, in any case, the inference time is kept below the allowed latency bound (*i.e.,* 20 s). In the same manner, finer-grained multi-container deployments also reduce the wait time of the samples, as shown in Figure 4.7 and Figure 4.8, which display the issue delay time of individual samples with *ANY* and *CPUMEM*, respectively. In particular, the plots show that the saturation point, *i.e.,*, when the samples start to wait resulting in some delay time, appears later for finer-grained multi-container deployments.

The better performance of the multi-container deployment schemes is a consequence of their ability to optimize the scheduling of the serving threads onto the available resources (processors and memory nodes), mainly by favouring processor affinity, which reduces context switches and migrations and memory affinity exploits data locality, thus

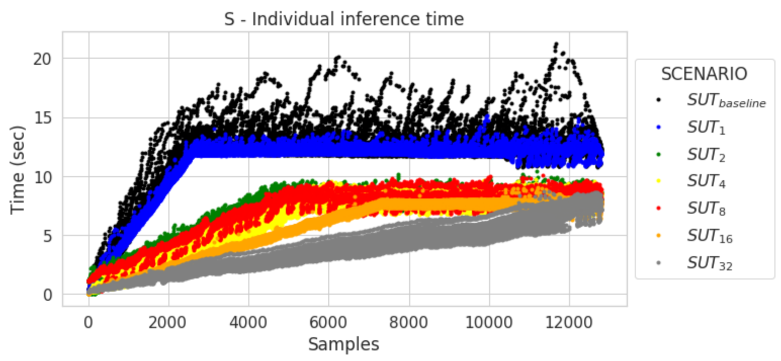


Figure 4.5: Inference time of individual samples in *Server-ANY* scenario.

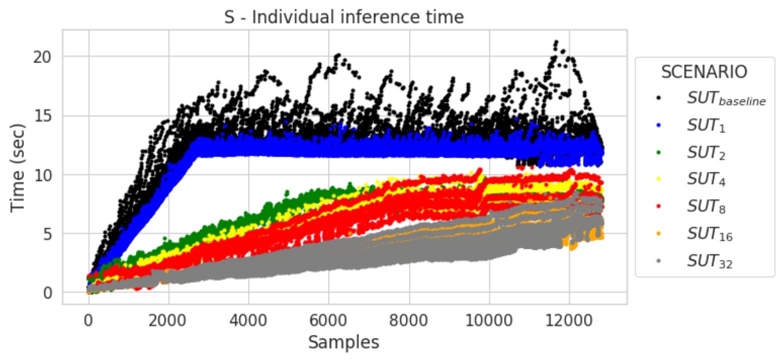


Figure 4.6: Inference time of individual samples in *Server-CPUMEM* scenario.

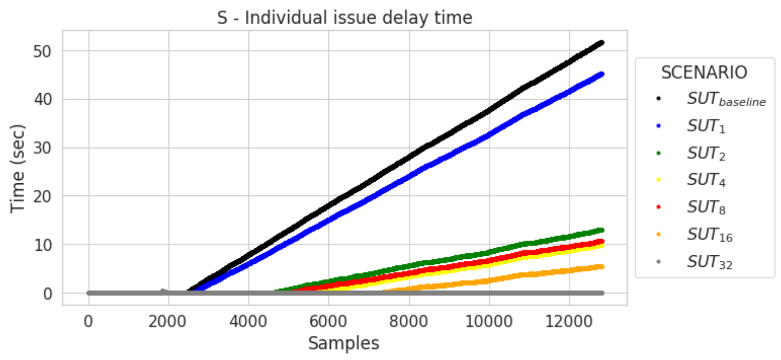


Figure 4.7: Issue delay time of individual samples in *Server-ANY* scenario.

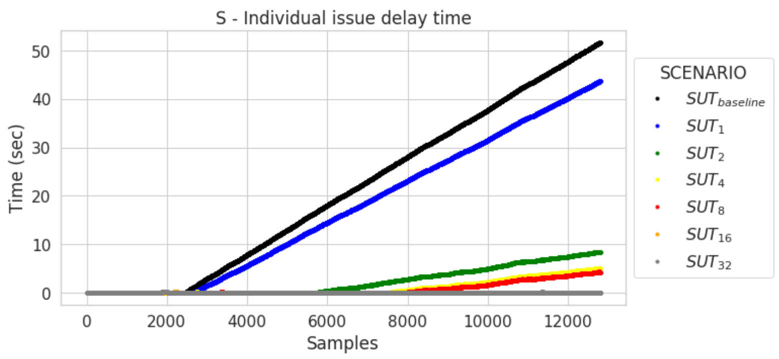


Figure 4.8: Issue delay time of individual samples in *Server-CPUMEM* scenario.

improving cache usage and reducing remote memory accesses in NUMA systems. With *CPUMEM* settings, the affinity for each container is enforced explicitly by the deployment scheme, which allocates dedicated CPUs to each of them. With finer-grain deployments, each container is allocated with fewer CPUs (and from a single NUMA node), thus there are fewer chances for the serving threads to migrate. With *ANY* settings, the affinity for each container is not enforced explicitly, but indirectly encouraged through the scheduling of cgroups done by the Linux Completely Fair Scheduler (CFS). The processes within each container are grouped together in a cgroup, so they will be viewed by the scheduler as a single unit. CFS applies the principle of sharing the resources fairly among these cgroups at the same level of the hierarchy, which means it will first divide CPU time equally between all entities at the same level, and then proceed by doing the same in the next level [46]. In multi-container deployment scenarios, first, the CPUs are evenly distributed across cgroups. Then, the threads on each cgroup are scheduled on those CPUs. As a higher number of containers contain a lower number of threads, this scheduling within the group is simpler, allowing to exploit processor affinity better. Notably, in SUT_{32}, the sole thread in each container runs on a single CPU, akin to being pinned to it.

ANY/CPUMEM affinity: Regarding the affinity settings, SUT_1 behaves similarly with both *ANY* and *CPUMEM* settings because the single container deployed in both cases uses the same range of CPUs and memory from the two sockets. SUT_2–SUT_{32} with *CPUMEM* settings show better performance than *ANY* up to 4%, 9%, and 3% in scenarios MS (see Figure 4.3), S (see Figure 4.2c), and O (see Figure 4.2d), respectively. This improvement also shows up when considering the mean latency. As shown in Figure 4.4, SUT_2–SUT_{32} with *CPUMEM* settings in *Server* scenario have 23%–29%–35%–58%–23% mean latency improvements, respectively, with respect to *ANY*. Similarly, SUT_2–SUT_{32} with *CPUMEM* settings in *Offline* scenario also show 3%–3%–3%–4%–1% improvements on the mean latency regarding *ANY*, as shown in Figure 4.9.

CPUMEM has better performance than *ANY* because it enforces CPU affinity, which restricts the number of assigned CPUs within each container. Hence, the threads running in finer-grained containers have fewer available CPUs where they could migrate, and more importantly, memory affinity, which improves the cache utilization and prevents as well the remote memory accesses, thus reducing the memory latency. Note that the improvement of SUT_{32} with *CPUMEM* regarding SUT_{32} with *ANY* is less noticeable than in the rest of SUT_2–SUT_{16} scenarios because in SUT_{32} each container runs only

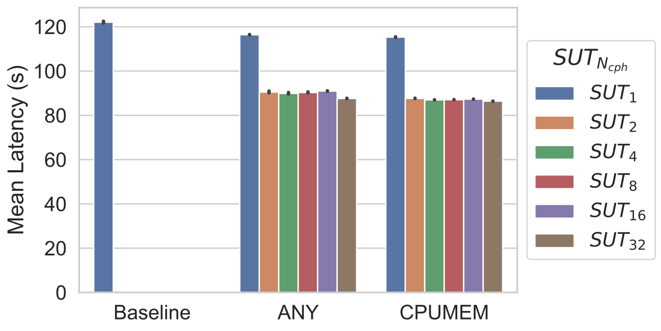
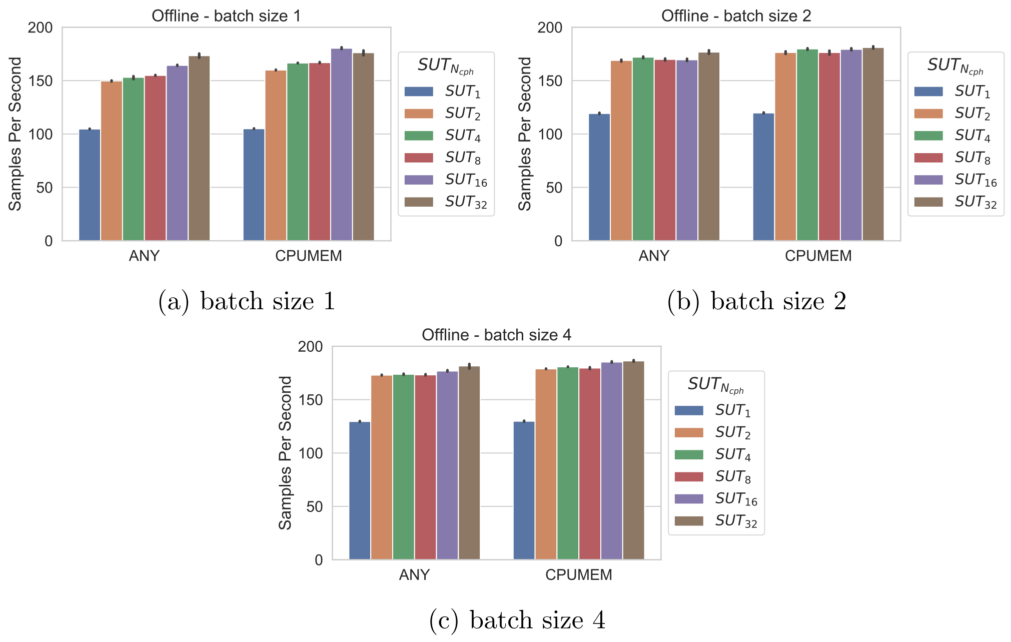


Figure 4.9: Mean latency in *Offline* ANY and CPUMEM scenarios.

on one CPU and containers are already well-distributed among cores, thus, the benefit of *CPUMEM* settings on SUT_{32} only comes from the memory access.

4.3.3 Multi-container Deployment Evaluation With Different Client Batch Size on a Single Host

Batching calls to a remote service is a well-known technique to increase the performance. There are fixed processing costs for any interaction with a remote service, such as serialization, network transfer, and deserialization. Packaging many samples into a single batch minimizes the cost per sample.



(a) batch size 1

(b) batch size 2

(c) batch size 4

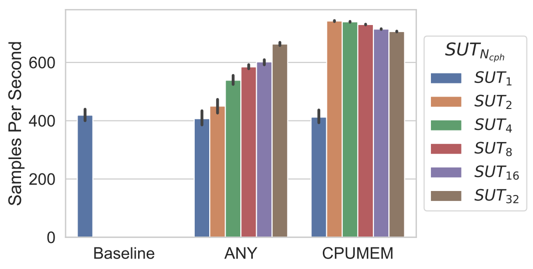
Figure 4.10: Impact of container granularity and affinity in the *Offline* scenario with different client batch size.

Figure 4.10 shows the impact of container granularity and affinity in the *Offline* scenario with various client batch sizes, namely 1, 2, and 4. By comparing this figure with Figure 4.2d, which set the batch size as 8, the overall performance is increased with larger batch sizes. For instance, the throughput of SUT_1 increases up to 26% from batch size 1 to 8.

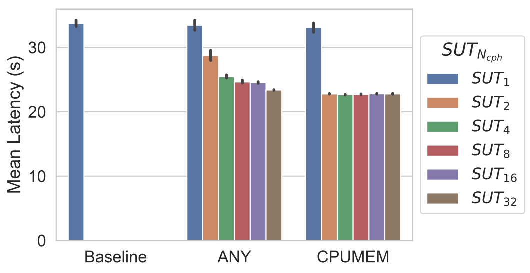
Regarding the impact of container granularity, Figure 4.10 shows that multi-container deployment schemes outperform the single container deployment for all the batch sizes. In particular, for batch size 1, SUT_2–SUT_{32} have 43%–47%–48%–57%–67% and 53%–59%–59%–72%–70% performance improvement regarding SUT_1 with *ANY* and *CPUMEM* settings; for batch size 2, SUT_2–SUT_{32} have 42%–45%–43%–43%–49% and 48%–51%–49%–50%–52% improvement regarding SUT_1 with *ANY* and *CPUMEM*; and for batch size 4, SUT_2–SUT_{32} have 33%–34%–33%–36%–41% and 38%–39%–39%–43%–44% improvement regarding SUT_1 with *ANY* and *CPUMEM*. Interestingly, smaller batch sizes can benefit more from multi-container deployments. As for affinity, *CPUMEM* also outperforms *ANY* for all the batch sizes, providing, again, a higher benefit for smaller ones. Notably, the throughput increases up to 10%–7%–6%–4% for batch sizes 1, 2, 4, and 8, respectively.

4.3.4 Multi-container Deployment and Affinity Evaluation on a Four-node Cluster

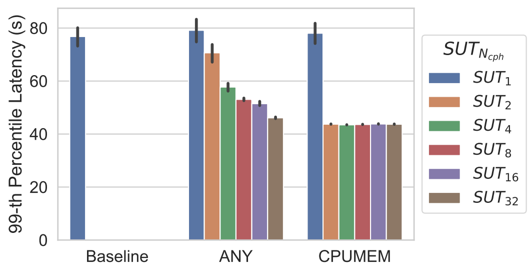
Experiments in the previous sections were run in a single node. Nevertheless, we anticipate that most of the performance insights obtained in those sections would still hold for multi-container deployment schemes in a larger cluster.



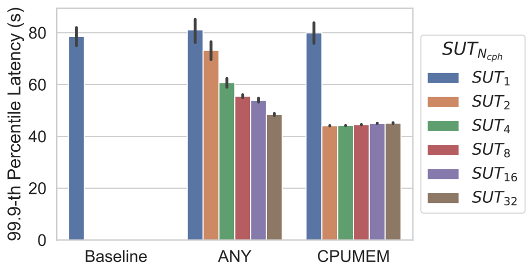
(a) Throughput



(b) Mean latency



(c) 99th tail latency



(d) 99.9th tail latency

Figure 4.11: Impact of container granularity and affinity in the *Offline* scenario at scale.

Figure 4.11a - Figure 4.11b show the impact of container granularity and affinity in the *Offline* scenario on a four-node cluster. SUT_2–SUT_{32} have 13%–35%–49%–55%–69% and 15%–24%–27%–27%–36% throughput and mean latency improvement, respectively, regarding SUT_1 with *ANY* settings. SUT_2–SUT_{32} have 87%–86%–84%–80%–78% and 32% (for all the SUT_i) throughput and mean latency improvement, respectively, regarding SUT_1 with *CPUMEM*. Latency improvements with *ANY* and *CPUMEM* are comparable, but throughput improvements are considerably higher with *CPUMEM* affinity, as it shows up to 68% improvement with respect to *ANY*. As anticipated, the performance observations and conclusions described in Section 4.3.2 also apply here.

Figure 4.2c - Figure 4.2d show the 99th and 99.9th tail latencies. For *ANY*, finer-grained containers have better tail latency. In particular, 99th latency of SUT_2–SUT_{32} improves 13%–28%–35%–37%–44% regarding SUT_1 and 99.9th latency of SUT_2–SUT_{32} improves 12%–26%–33%–36%–42% regarding SUT_1. As shown in Figure 4.12, which displays the individual inference time of each sample at *ANY*, the last samples up to 3% show a tail latency increase in all the scenarios. This is because the tail is less CPU-intensive and all the containers are about to finish their tasks. When tasks in one container (*i.e.,* one cgroup) become idle and are not using any CPU time, the leftover time is collected in a global pool of CPU cycles that can be used by other containers (*i.e.,* other cgroups) from this pool. Finer-grained deployments show a better tail latency because they have more cgroups, thus each container releases fewer CPU cycles when it finishes, causing fewer CPU migrations for the rest of running containers.

For *CPUMEM*, multi-container deployments show up to 47% and 46% improvement on 99th and 99.9th latency, respectively, regarding SUT_1, but there is a minor difference among the various multi-container schemes on 99th latency (less than 1%) and 99.9th latency (2% difference). Multi-container deployments in *CPUMEM* show almost no overhead in the tail latency because each container has its own cgroup without the CPUs overlap.

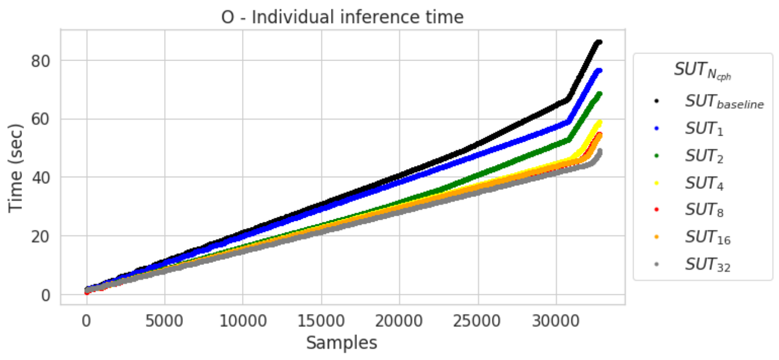


Figure 4.12: Inference time of individual samples in *Offline-ANY* scenario.

4.4 Conclusion and Future Work

This chapter presented multi-container deployment schemes for containerized online ML inference services on Kubernetes. We focused on the container layer to understand how the container granularity and its combination with CPU/memory affinity impact the performance of online ML inference services. We concluded that multi-container deployments show significant performance improvements up to 69% and 87% regarding the single-container deployment on single-node and four-node clusters, respectively. Finer-grained deployments show better performance because they favour process affinity in a similar way to when threads are pinned explicitly. Consequently, these deployments fit very well with explicit CPU/memory affinity settings for each container. As demonstrated in our experiments, those settings can sum up to 9% and 68% to the granularity gains on single-node and four-node clusters, respectively. The benefit of multi-container deployment schemes with affinity also shows up with different client batch sizes and in larger clusters.

All in all, we demonstrated that it is worth considering (and optimizing) the containerization dimension when provisioning ML inference services to benefit not only from its encapsulation, security, and fault isolation, but also gain performance. Moreover, the granularity/affinity settings at the container-level are complimentary to other optimizations such as batching and autoscaling and, therefore, can be combined to derive better deployment and scheduling policies for ML inference services.

In the future, we will consider the performance insights in this paper about the container-level settings (*i.e.,* container granularity and affinity) to derive placement policies integrated within the Kubernetes scheduler/Kubelet agent for the efficient deployment of online multi-model ML inference services in a multi-programmed and multi-tenant Cloud environment.

Chapter 5

Scanflow-Kubernetes: Agent-based Framework for Autonomic Management and Supervision of ML Workflows in Kubernetes Clusters

This chapter is based on the work done in collaboration with Lenovo Infrastructure Solutions Group, which has resulted in a demo paper, a full paper at CCGRID conference, and the open-source platform Scanflow-Kubernetes:

[**1**] Peini Liu, Gusseppe Bravo-Rocca, Jordi Guitart, Ajay Dholakia, David Ellison, and Miroslav Hodak, "Scanflow: an end-to-end agent-based autonomic ML workflow manager for clusters," *In Proceedings of the 22nd International Middleware Conference: Demos and Posters*, December 2021, Virtual Event, Canada. pp. 1-2, DOI: 10.1145/3491086.3492468. (CORE RANK A)

[**2**] Peini Liu, Gusseppe Bravo-Rocca, Jordi Guitart, Ajay Dholakia, David Ellison, and Miroslav Hodak, "Scanflow-K8s: Agent-based Framework for Autonomic Management and Supervision of ML Workflows in Kubernetes Clusters", *2022 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2022, Taormina, Italy, pp. 376-385, DOI: 10.1109/CCGrid54584.2022.00047. (CORE RANK A)

[**3**] May 29, 2021 - Software Release on the Github repository: "Scanflow-Kubernetes: An MLOps Platform". Available at: `https://github.com/bsc-scanflow/scanflow`.

- M1: (26/06/2021) Release master v0.1.0 Scanflow-Kubernetes basic.
- M2: (13/12/2021) Release master v0.1.1 Scanflow-Kubernetes with resource, affinity and HPA definition.

In this chapter, Section 5.1 presents a brief introduction of machine learning workflows and the need to do autonomic management. Section 5.2 introduces the multiple management layers in autonomic ML workflows. Section 5.3 describes the agent architecture, social ability, triggers, and operation primitives. The detailed Scanflow-K8s implementation is described in Section 5.4. Section 5.5 presents some case studies and experiments on Scanflow-K8s platform. Finally, conclusion and future work are discussed in Section 5.6.

5.1 Introduction

Machine Learning (ML) has become common with good results in different tasks such as image classification, machine translation, recommendation systems, and speech recognition. While working on a ML project, workflows comprising some reproducible steps run as a pipeline are widely used to build or deploy a model efficiently because of the flexibility, portability, and fast delivery they provide to the ML life-cycle [48].

ML workflows still face several challenges while being used by different teams. The Data Science team requires to automate some repetitive tasks within ML workflows to train and improve the model [199][155]. Therefore, some AutoML modules and frameworks have been developed for algorithm selection [19], model selection [38], and feature selection [77] to tune hyperparameters and have good learning performance with less human assistance. However, ML life-cycle is more than just training a model [49]. Once the model has been trained, the Data Engineer team works on deploying the ML workflows into production. More importantly, they are required to operate the workflows to maintain the robustness of the model at runtime, that is, to deal with security vulnerabilities, concept drift, lack of explainability and interpretability, and hidden technical debt [42][106], because the model may degrade its accuracy due to constantly evolving data profiles. Also, the model online inference serving services have strict latency requirements and efficiency issues that should be considered [134][2][27]. Therefore, the ML workflow is no longer running in a known context and with static requirements, and consequently, enabling the autonomy to manage and supervise ML workflows to meet dynamic changes has become an open issue [78].

Autonomic computing brings inspiring approaches to adapt ML systems at runtime, helping to manage and supervise the ML workflows operation in dynamic contexts [103][30][177][21]. For example, by enabling adaptive learning algorithms for streaming data to supervise ML models at the application layer or by reconfiguring and restructuring the workflows at the infrastructure layer [211]. Consequently, our work enables an agent-based approach to leverage autonomic computing for ML workflows system to meet dynamic changes. The agents focus on the robustness and requirements of the model at the application layer while managing the quality of services and the structure of workflows at the infrastructure layer.

In this chapter, we contribute Scanflow-K8s, a functional agent-based MLOps framework that enables autonomic management and online supervision of the end-to-end lifecycle of ML workflows on Kubernetes. Scanflow-K8s redesigns Scanflow from scratch to upgrade the executor nodes to a multi-agent system based on triggers, primitives, and strategies, and to be fully integrated with the Kubernetes platform, enabling autonomic multi-layer management and supervision of ML workflows in clusters.

5.2 Architecture for Autonomic ML Workflows

In this section, we firstly describe diverse uncertainties that occur in ML workflows. Then, we present an architecture for autonomic ML workflows featuring a multi-layered autonomic framework. Finally, we present a practically implemented platform that enables autonomic ML workflows on Kubernetes clusters based on agents.

5.2.1 Uncertainties in ML Workflows

The need to embed ML systems into long-lived dynamic contexts is likely to increase in the coming years [78], thus inherent uncertainties in ML workflows may increase when they are deployed in production (*e.g.,* Cloud). Table 5.1 shows a taxonomy of potential uncertainties in ML workflows.

Uncertainties in ML workflows come from two main sources, namely the requirements and the context. The former comes from the data scientists and end-users and includes the functional and non-functional requirements of the ML workflows. In particular, (1) the functional requirements can include changes in the topology of the ML workflows (*e.g.,* adding new steps), as well as the need to maintain the quality and robustness of the ML models (*e.g.,* dealing with security vulnerabilities, outliers, concept drift, and model explainability) in unexpected situations; (2) the non-functional requirements can include

Table 5.1: Uncertainties in ML workflows

Categories		**Examples**
Requirements	Functional Requirements	End-users expect robust ML models when facing data drift; Data scientist adds new steps to the workflow
	Non-functional Requirements	End-users define some QoS requirement for model serving; Data engineer provides resource restrictions and affinity settings for workflow executors
Contexts	Workflow Contexts	Workflow executor fails to run (*e.g.,* software bug, out-of-memory error); Model serving service is not available
	System Contexts	Other workflows compete to use the same resources
	External Contexts	Hardware/Operating System crashes or is not available

the need to fulfill the QoS (Quality of Service) guarantees related to the model serving service's performance (*e.g.,* latency and failure rate) in the occurrence of churn, as well as the definition of runtime parameters (*e.g.,* resource restrictions and affinity settings) for workflow executors and services. The uncertainties in the context come from the execution of the workflows themselves, their interaction with other workflows, and the software/hardware platform. In particular, (1) the changes in the workflows contexts happen in workflows themselves, for instance, a workflow executor fails (*e.g.,* software bug, out-of-memory error), or a workflow service is not available; (2) the changes in the internal system contexts derive from the relationship between the various workflows (and other applications) and how the orchestrator arbitrates their use of the shared platform where they run, for instance, when the resources needed to run a workflow may be in use by other workflows; (3) the changes in the external contexts occur in the underlying platform, including hardware resources, operating systems, and other related systems, which can fail or become unavailable. Those changes can be detected and the manager can react to them but cannot be directly solved at the management level.

5.2.2 Multi-layered Control for Autonomic ML Workflows

Each type of uncertainty requires applying different strategies to enable autonomy for ML workflows. These strategies could reside at different layers, for instance, ML workflows could react to changes by restructuring the workflow topology or by reconfiguring the workflow executor/service instances. Thus, we should implement controllers at several layers to manage ML workflows in a completely autonomic way.

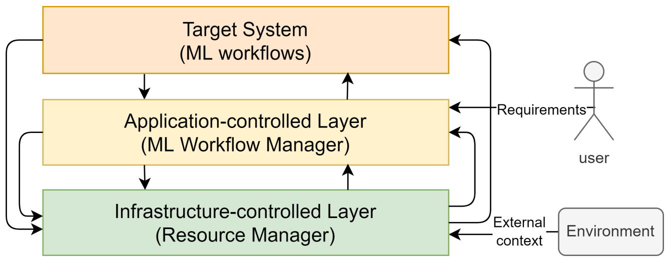


Figure 5.1: Conceptual architecture of multi-layer controlled ML workflows.

A conceptual architecture for multi-layer controlled autonomic ML workflows is proposed in Figure 5.1, which shows the layers and their interactions. ML workflows are the target system focusing on the ML business. From a static design perspective, ML workflows are composed of some reproducible steps and organized by dependencies. From a dynamic implementation perspective, ML workflows could be run as containerized executors in a pipeline or deployed as online services consisting of microservice instances. The management system actuates on multiple control layers, namely the application-controlled layer and the infrastructure-controlled layer, which can operate the target system to deal with different types of uncertainty: (1) the application-controlled layer senses the application-related changes, such as the requirements from data scientists and end-users and the workflow context, and restructures the static view of the target system, which is executed with the help from the lower control loops; (2) the infrastructure-controlled layer senses the internal and external system contexts, and adjusts the workflow executors or services at run-time accordingly to the predefined rules of the resource manager by taking advantage of the orchestrator resource management capabilities. In this case, the system autonomy appears as a form of reconfiguration.

5.2.3 A Practical Platform for Autonomic ML Workflows

This section describes Scanflow-K8s, a practical platform for autonomic ML workflows that implements the above-mentioned multi-layered control autonomy by means of the integration of a ML workflow manager (*i.e.,* Scanflow) with an orchestrator (*i.e.,* Kubernetes). The whole architecture of this platform is depicted in Figure 5.2.

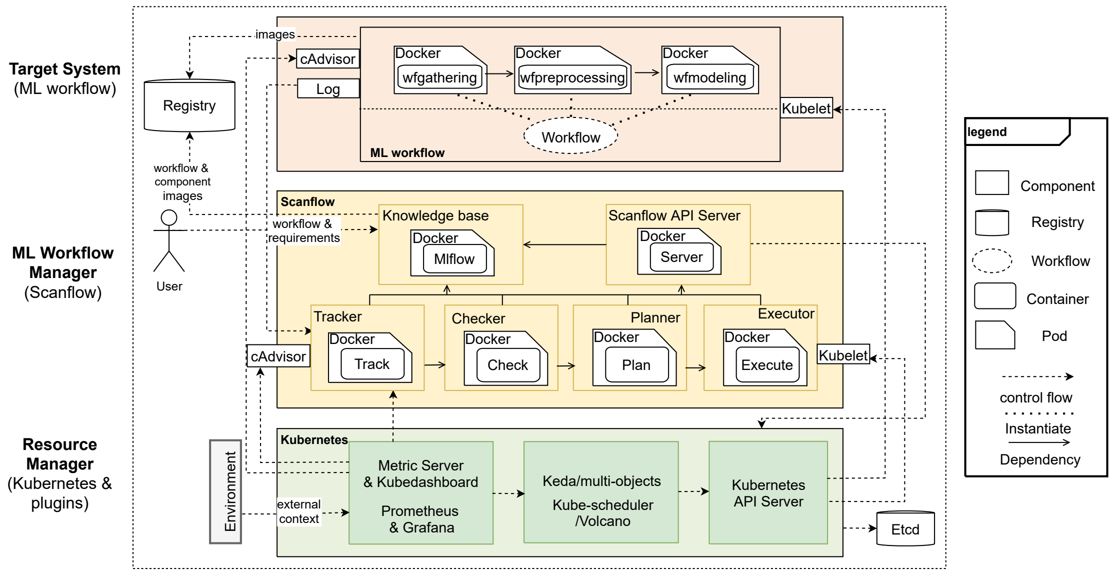


Figure 5.2: Scanflow-K8s: A practical platform for autonomic ML workflows.

Target System: The top of the figure shows the ML workflows which are the target system focusing on the ML business. From a static design perspective, ML workflows define some steps and their dependencies. As shown in Figure 5.3, two types of workflows are supported by Scanflow-K8s for both training and inference phases. On the left side, a ML workflow is defined as a batch pipeline composed of several executors E_i that are executed in sequence or in parallel. In a batch inference, predictions can be generated asynchronously with a batch of samples and the time to get the results is unconstrained. On the right side, a ML workflow is defined as online services with graph traffic forwarding. In an online inference, predictions are served in real time, typically subject to a latency bound. From a dynamic implementation perspective, the batch executors or the online services are conducted as containerized instances executing locally or in the Cloud. In particular, the executors of batch workflows run once for each time the workflow is executed, and the online workflow is deployed as a long-run microservice that is able to deal with clients' invocations. Normally, the Data Science team uses the batch ML workflows to build and gain ML models at the ML training phase, while the Data Engineer team conducts the batch ML workflows for batch predictions or deploys online ML workflows in production to make real-time predictions at the ML inference phase.

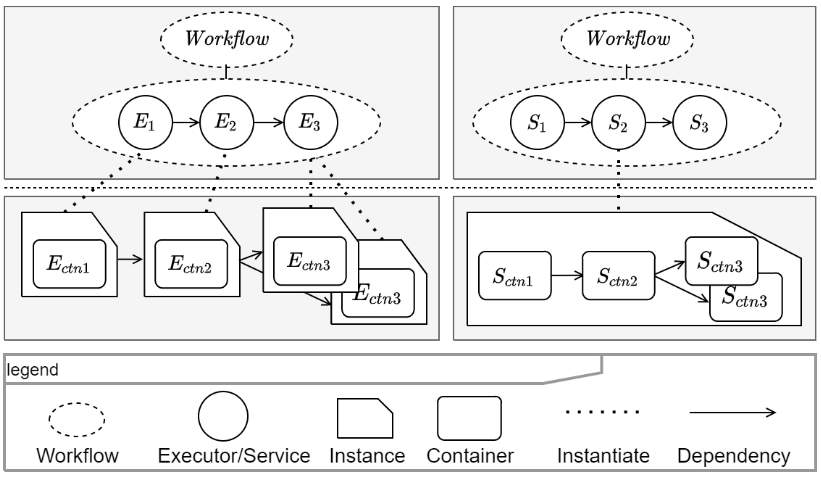


Figure 5.3: ML workflows supported by Scanflow-K8s (a batch ML workflow on the left and an online ML workflow on the right).

Application-controlled Layer: ML workflow manager (*i.e.,* Scanflow) is used as a controller of the application layer, as shown in the middle of Figure 5.2. Scanflow is composed of multiple reactive agents, which work together to perform adjustments in ML workflows to deal with application-related changes. Internally, Scanflow supports four predefined agent templates, namely tracker, checker, planner, and executor. A tracker-agent, which is based on Mlflow[1], is used to collect the metrics (*e.g.,* number of predictions) or logs (*e.g.,* prediction results) from ML workflows and save information in a knowledge base. A checker-agent can define thresholds to detect outliers or use learning methods to check drift anomalies, which are both based on real-time stream executions and knowledge from a tracker-agent. A planner-agent can decide how to address the detected issues, for instance by retraining the model using transfer learning to improve its robustness based on knowledge from tracker-agent and checker-agent. Finally, the operating plans from a planner-agent can be organized as a set of actions, for example upgrading or changing the version of the model, which are then carried out by an executor-agent, which manages the application-layer internal changes, and the Scanflow API server, which communicates with the infrastructure layer to adjust the

[1]`https://www.mlflow.org/docs/latest/tracking.html#scenario-4-mlflow-with-remote-tracking-server-backend-and-artifact-stores`

target system.

Infrastructure-controlled Layer: The bottom of Figure 5.2 shows the resource manager working on the infrastructure-controlled layer. Scanflow's best practice is integrating with the well-known Kubernetes orchestrator, given that our ML workflows are wrapped as containers that can be finely managed, and Scanflow-K8s can take advantage of the wide range of toolkits in the Kubernetes ecosystem. Used toolkits are presented in Table 5.2.

Table 5.2: Kubernetes toolkits.

Tool	**Role**
Kubernetes[2]	Container orchestration, automated container deployment, scaling, and management.
Istio[3]	Service-to-service connection and traffic monitoring.
Prometheus[4]	Metrics monitoring and alerting.
Volcano[5]	Batch workflow scheduling.
Keda[6]	Event-driven autoscaler.
Argo Workflows[7]	Multi-step workflow engine supporting DAG.
Seldon Core[8]	Online model serving on Kubernetes.

The infrastructure-controlled layer supports the deployment and execution of our containerized ML workflows on the platform by leveraging Kubernetes. At the training phase, ML workflows are defined as batch executors and are executed in K8s as Argo Workflows. At the inference phase, ML workflows can be defined both as batch executors or online services, according to data engineers' preferences. The former are executed as Argo Workflows (as in the training phase), whereas the latter are deployed and executed using Seldon.

At the infrastructure-controlled layer, the resource manager senses the system and external contexts from the environment, and enables autonomic ML workflows by performing finer-grain adjustments at run-time. For the monitoring, Kubernetes internal metric server and Prometheus toolkit collect the status of the cluster and the performance/resource usage of ML workflows executors or service instances. Also, Istio service mesh traces the traffic and security of each invocation. For the analysis and optimization, the manager can choose the optimal values for the configurable thresholds, which will be used by the HPA (Horizontal Pod Autoscaler) or Keda autoscaler to decide the number of instances, as well as configure the scheduling policy for the default kube-sheduler and the batch scheduler Volcano, which will be used to decide the allocation of ML workflows. Finally, the decided actions are carried out by the Kubernetes API server, which hands out the operations to the kubelet within the cluster to adjust ML workflows in order to adapt to the changing context.

Moreover, the ML workflow manager can govern some changes in collaboration with the resource manager. For example, some application-related run-time information at

[2] https://kubernetes.io/
[3] https://istio.io/
[4] https://prometheus.io/
[5] https://volcano.sh/en/
[6] https://keda.sh/
[7] https://argoproj.github.io/
[8] https://www.seldon.io/

the infrastructure layer can be tracked by the agents and considered at the application layer. Similarly, some application-related changes in the requirements/decisions need to be implemented in the infrastructure layer, which requires the Scanflow API server to communicate with Kubernetes, for example, to autoconfigure the application thresholds in Keda autoscaler or the affinity/resource limits of workflows according to the user's requirements, and to operate workflows in case of a fail-over to a user-defined backup service.

5.3 Agents for Autonomic ML Workflows

In this section, we introduce the architecture of Scanflow agents and their features. In detail, we define the agent communication, triggers, and operation primitives for ML workflows under uncertainties.

5.3.1 Agent Architecture

We use the concept of reactive agent, which does not implement a global model or plan but only some simple behaviors. These behaviors allow the agent to react when the environment changes. An agent includes a sensor that senses internal and external state changes, a set of conditional rules that respond to related events, and an actuator that activates a certain process of the environment or other agents.

Scanflow agents are the fundamental components to implement autonomic ML workflows. Each agent is an independent computational unit that is able to run actions according to the state changes. Therefore, an agent can be defined as a set of state-to-action mappings (*i.e.,* $Agent = States(s) \rightarrow Actions(a)$), that is, state changes could result in the execution of actions (if the conditional rules are satisfied). However, an agent usually cannot directly perceive the states but compute them from observations o_t using a function F. Also, the agent performs actions through rules with the computed states s_t ($a_t = R(s_t)$). Figure 5.4 shows the agent-environment interaction: At time t, the agent computes the states s_t from the observations o_t using a function F. Then, it chooses actions a_t according to rules R to achieve the agent's goal.

To cope with the autonomic management and online supervision for ML workflows, each agent implements its autonomy by defining strategies that include events, constraints, and actions. The autonomic management strategy represents the automation scenarios and can be expressed as 3-tuples $Strategy = (Events, Constraints, Actions)$, where an *Event* is mainly a state change, which is judged from the observations gathered by the agent triggers, a *Constraint* is a boolean expression, which refers to whether an attribute value fulfills a condition (*e.g.,* fitting a threshold), and an *Action* is a single or combined operation primitives or a request to call other agents. Specifically, the autonomic management strategy of the agent is described as: when the *Event* happens, if the *Constraint* is satisfied, then the *Action* will be executed.

5.3.2 Agent Social Ability

Social ability describes how multiple agents could collaborate to solve problems by interacting with each other. Traditionally, interaction has been modeled through agent communication languages, such as FIPA-ACL[9]. Recently, researchers have proposed other

[9] http://www.fipa.org/

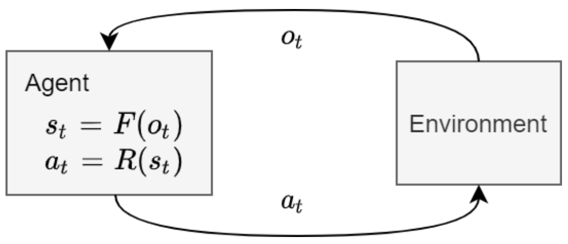


Figure 5.4: Agent-Environment interaction.

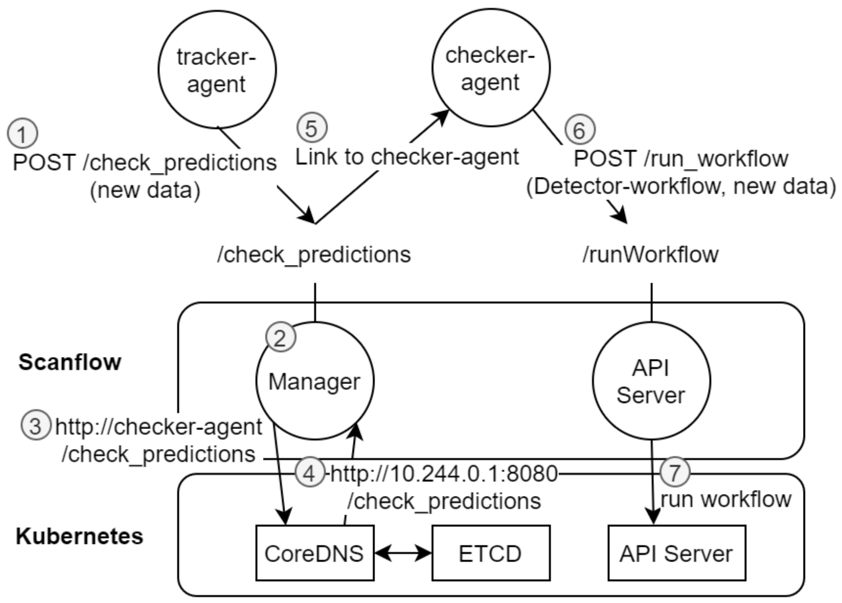
interaction methods based on concepts like using a shared volume [20]. Scanflow leverages microservice-based agents [187] which could also interact with each other transparently with a service discovery through RESTful APIs. In practice, a single approach of social ability is often insufficient, and thus Scanflow agents apply both shared artifacts and RESTful APIs communication approaches to support the social ability of agents.

- Interaction through RESTful APIs: In this approach, the states or actions of an agent are exposed as interfaces. Agents need to be registered first into a service discovery, then they could call the well-defined interfaces from other agents through REST. Normally, the remote call leads to changing the belief/state of the agent and will finally drive an action. Figure 5.5 exemplifies how Scanflow agents communicate with RESTful APIs. Tracker-agent asks for an agent to check for anomalies in the predicted new data. First, tracker-agent needs to specify which action it wants (*e.g.,* check_predictions); then Scanflow manager will generate the service domain name of the agent and request a specific IP address by using CoreDNS, which resolves the domain name, and Etcd, which returns the IP address from a service name. Thus, tracker-agent could finally link to the checker-agent. This RESTful POST from the tracker-agent changes the state of the checker-agent, therefore, the checker-agent will POST a run_workflow action to Scanflow API server and Kubernetes API server to carry out its belief (*e.g.,* run detector workflow to check the anomaly of predicted new data).

- Interaction through shared artifacts: This approach communicates through shared artifacts within an application-related knowledge base which receives queries from agents and delivers the results from its database. These include the metadata and logs from the prediction service, and the metrics, scores, parameters, and different versions of the ML model. The states of Scanflow agents can be easily updated through RESTful interaction, but, for complex operations with large data involved, it is more efficient to use shared artifacts so that agents could make actions directly with the accessible resources.

5.3.3 Agent Triggers

To actively monitor current *States*, agents are required to trigger tasks to sense the useful observations. Scanflow provides different types of built-in triggers, namely interval triggers, date triggers, and cron triggers (see Table 5.3). Also, the basic triggers can be combined together using ‘and’ or ‘or’ logic to produce more complex hybrid triggers. These triggers can be scheduled at a specific time or time intervals to execute tasks so that agents could get required observations to evaluate the changes of *States*. Note that



tracker-agent
checker-agent
1 POST /check_predictions (new data)
5 Link to checker-agent
6 POST /run_workflow (Detector-workflow, new data)
/check_predictions
/runWorkflow
Scanflow
2 Manager
API Server
3 http://checker-agent /check_predictions
4 http://10.244.0.1:8080 /check_predictions
7 run workflow
Kubernetes
CoreDNS
ETCD
API Server

Figure 5.5: Agents communicate with RESTful APIs.

each Scanflow agent contains an asynchronous I/O scheduler with multiple queued tasks. Tasks are run by the scheduler in a thread pool.

Table 5.3: Types of agent triggers.

Types		Definition
Scheduled Triggers	Interval	Trigger at the specified frequency.
	Date	Trigger once on the given date and time.
	Cron	Trigger when current time matches all specified time constraints (similarly to UNIX cron).
Action Triggers	Call-Receive	Call-Receive interface for agent to be triggered through invocations from other agents.

On the other hand, an agent can also be triggered by external actions. For example, receiving invocations from other agents, as discussed in Section 5.3.2.

5.3.4 Operation Primitives

After some change in the *States*, the agents need to perform *Actions* (*i.e.*, $a_t = R(s_t)$). Therefore, we propose some operation primitives that represent the atomic autonomic management steps. The execution of a single primitive or a series of combined primitives is able to implement a full action of an agent. Given that Scanflow agents can manage the ML system by making adjustments both at the application layer and the infrastructure layer (as described in Section 5.2.3), and that both batch and online workflows should be supported, the primitive operations should be designed to happen at those layers and to adapt to those types of workflows.

As shown in Table 5.4, at the application layer, we propose primitives for both types of ML workflows to manage the ML workflow itself, and to set requirements (*e.g.*, affinity, resource limits, etc.) for the workflow from the users' perspective. At the infrastructure

Table 5.4: Agent operation primitives.

	Application layer	**Infrastructure layer**

Batch ML workflow	runWorkflow()
	stopWorkflow()
	upgradeWorkflow()
	updateWorkflowAffinity()
	updateWorkflowResource()
		runExecutor()
		stopExecutor()
		upgradeExecutor()
		updateExecutorAffinity()
		updateExecutorResource()

Online ML workflow	deployWorkflow()
	deleteWorkflow()
	upgradeWorkflow()
	updateWorkflowAffinity()
	updateWorkflowResource()
	updateWorkflowReplica()
	updateWorkflowTraffic()
		applyWorkflowInstance()
		deleteWorkflowInstance()
		duplicateWorkflowInstance()

layer, we introduce primitives for operating executors of batch ML workflows or instances of online ML workflows in order to collaborate with the resource manager.

Regarding batch ML workflows, from the application layer, the agents can control the life cycle of the workflow and update its metadata, parameters, and artifacts. For example, the planner-agent can restart the training workflow to retrain the model through *runWorkflow*(); the executor-agent can update the version of the workflow ML model by using *upgradeWorkflow*() and can update the affinity (using *updateWorkflowAffinity*()) or resource limits requirements (using *updateWorkflowResource*()) with the knowledge from the planner-agent. From the infrastructure layer perspective, the executors will be run and guaranteed by the resource manager, but the agents can actively run or stop an executor using *runExecutor*() or *stopExecutor*(), respectively. For example, replicated executors can be stopped in case any one of them has finished the task. Also, a specific executor within the workflow can be upgraded, for instance, the planner-agent can update the input parameters for the data-gathering executor of the workflow by using *upgradeExecutor*() and also change its run-time settings by using *updateExecutorX*() operations.

Regarding online ML workflows, from the application layer, the agents can add, upgrade, delete, and update the microservice using *deployWorkflow*(), *upgradeWorkflow*(), *deleteWorkflow*(), and *updateWorkflowX*(), respectively. For example, when the model serving service in the workflow needs a new version of the model, the executor-agent must upgrade the microservice by using *upgradeWorkflow*(). The agents can also provide user's requirements to define application-related thresholds. For instance, the planner-agent may call *updateWorkflowReplica*() to set a failure rate or throughput threshold, so that the online ML workflow microservice will be scaled when the observed value is over the threshold. As for the infrastructure layer, the agents have the option to directly control the number of workflow serving instances. For instance, the executor-agent can call *duplicateWorkflowInstance*() to scale up and down the online ML workflow service.

5.4 Scanflow-K8s Platform Implementation

This section provides in detail the Scanflow-K8s implementation. It originally supports deploying and operating ML workflows on Kubernetes, but users can also extend it to other platforms. It is now open-sourced at Github `https://github.com/bsc-scanflow/scanflow`.

5.4.1 Scanflow-K8s Concepts

At the application level, Scanflow-K8s re-implements from scratch the Scanflow framework. It provides a high-level library that supports defining workflows, building each node of workflows and agents, and deploying/running the agents/workflows. In particular, it announces a framework for developing agents in order to manage and supervise workflows in both the ML training stage and the ML inference stage.

The features of Scanflow include:

- **Scanflow Developing** (Scanflow Application): A format for teams to define workflows, agents, and basic environment.
- **Scanflow Building**: To build Scanflow Application locally (each node of a workflow and agents as an image) and save images to repository.
- **Scanflow Deploying** (Scanflow Server): An API to create a working environment for each team and deploy agents. It also supports deploying workflows running as batch workflows or deploying them as online services.
- **Scanflow Operating** (Scanflow Agent): A framework to develop agents. Provides an online multi-agent system to manage and supervise the workflows.
- **Scanflow Tracking** (Supported by MLflow): MLflow provides an API to log parameters, artifacts, and models in machine learning experiments. We use MLflow as a database to track this information and transmit the information between teams.

5.4.2 Scanflow-K8s Components

Scanflow-K8s components are shown in Figure 5.6. The main components are Scanflow-server and Scanflow-tracker: the former is used for deploying Scanflow applications and Scanflow agents, the latter is used as a central data of Scanflow to be shared among teams/agents.

Scanflow also provides several clients to make use of the Scanflow-K8s platform. For instance: Scanflow provides a format for teams to define workflows, agents, and basic environments, therefore, users could utilize ScanflowClient to define their applications and directly build the images locally and save the images to the image repository (see Listing 5.1). Also, ScanflowDeployClient can be used to connect with Scanflow-server to deploy/run/terminate Scanflow applications on demand (see Listing 5.2). In case of some local information from users should be known among teams, ScanflowTrackerClient provides a way to connect with Scanflow-tracker directly to save some useful information (see Listing 5.3).

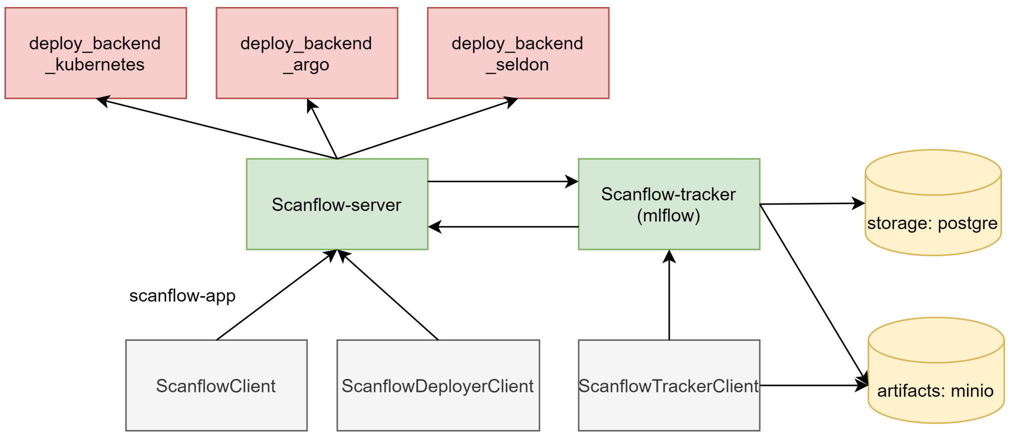


Figure 5.6: Scanflow-K8s Components.

```
import scanflow
from scanflow.client import ScanflowClient
# scanflow client
client = ScanflowClient(scanflow_server_uri='http://172.30.0.50:46666',
                                  verbose=False)
# define application
executor1 = client.ScanflowExecutor(name='load-data',
                      mainfile='loaddata.py',
                      parameters={'app_name': app_name,
                                  'team_name': 'data'})
executor2 = client.ScanflowExecutor(name='modeling-cnn1',
                      mainfile='modeling.py',
                      parameters={'model_name': 'mnist_cnn',
                                  'epochs': 1,
                                  'x_train_path': '/workflow/load-data/
    mnist/data/mnist/train_images.npy',
                                  'y_train_path': '/workflow/load-data/
    mnist/data/mnist/train_labels.npy',
                                  'x_test_path': '/workflow/load-data/
    mnist/data/mnist/test_images.npy',
                                  'y_test_path': '/workflow/load-data/
    mnist/data/mnist/test_labels.npy'},
                      requirements='req_modeling.txt')
dependency1 = client.ScanflowDependency(dependee='load-data',
                                    depender='modeling-cnn1')
workflow1 = client.ScanflowWorkflow(name='mnist-wf',
                     nodes=[executor1, executor2],
                     edges=[dependency1],
                     type = "batch",
                     output_dir = "/workflow")
app = client.ScanflowApplication(app_name = app_name,
                                 app_dir = app_dir,
                                 team_name = team_name,
                                 workflows=[workflow1])
# build application
build_app = client.build_ScanflowApplication(app = app, trackerPort
    =46668)
```

Listing 5.1: Usage of ScanflowClient.

```python
import scanflow
from scanflow.client import ScanflowDeployerClient
# scanflow deploy client
deployerClient = ScanflowDeployerClient(user_type="local",
                                        deployer="argo" k8s_config_file="
    /gpfs/bsc_home/xpliu/.kube/config",
                                       verbose=False)
deployerClient.create_environment(app=build_app)
deployerClient.run_app(app=build_app)
deployerClient.delete_app(app=build_app)
```

Listing 5.2: Usage of ScanflowDeployClient.

```python
import scanflow
from scanflow.client import ScanflowTrackerClient
# scanflow tracker client
trackerClient = ScanflowTrackerClient(
                        scanflow_tracker_local_uri="http
    ://172.30.0.50:46668",
                        verbose=False)
trackerClient.save_app_meta(build_app)
trackerClient.save_app_model(app_name=app_name,
                            team_name=team_name,
                            model_name="mnist_cnn")
```

Listing 5.3: Usage of ScanflowTrackerClient.

5.4.3 Scanflow-K8s for MLOps

Scanflow-K8s is a platform that provides features to simplify MLOps. The architecture of Scanflow-K8s for MLOps is shown in Figure 5.7. There are many phases and steps required to make the ML model in production to provide values.

The top of the figure shows the steps for the data team and data science team before a model runs into production. Normally, the data team is responsible for discovering and collecting valuable data, and the data science team will then develop a ML workflow that contains data preparation, validation, and preprocessing, as well as model training, validation, and testing. Workflow manager (*e.g., Scanflow*) can track the metadata such as metrics and scores and the artifacts during the training phase, analyze them, and automatically tune the hyper-parameters, early stopping and do neural architecture search for improving the model.

The bottom of the figure shows the model in production, including the model inference workflow deployment and the operation phase that automatically manages the ML workflow from both the application layer (*e.g.,* workflow manager Scanflow) and the infrastructure layer (*e.g.,* resource manager Kubernetes).

For deploying and managing the ML workflow at scale, the data engineer team should build a workflow managed by the workflow manager but wrap and deploy the model as a service. From the application-layer controlled view, the workflow manager could log the model metrics (such as scores) and artifacts (such as new data) to detect outliers, adversarial or drift and provide model explanations and finally trigger the ML workflow to be retrained or the model to be updated. From the infrastructure-layer controlled view, allowing the model as a service helps it to be released, updated and rolled out independently, and can monitor the latency and failure rate of its predicted invocations at inference time. With these observations, the resource manager can automatically

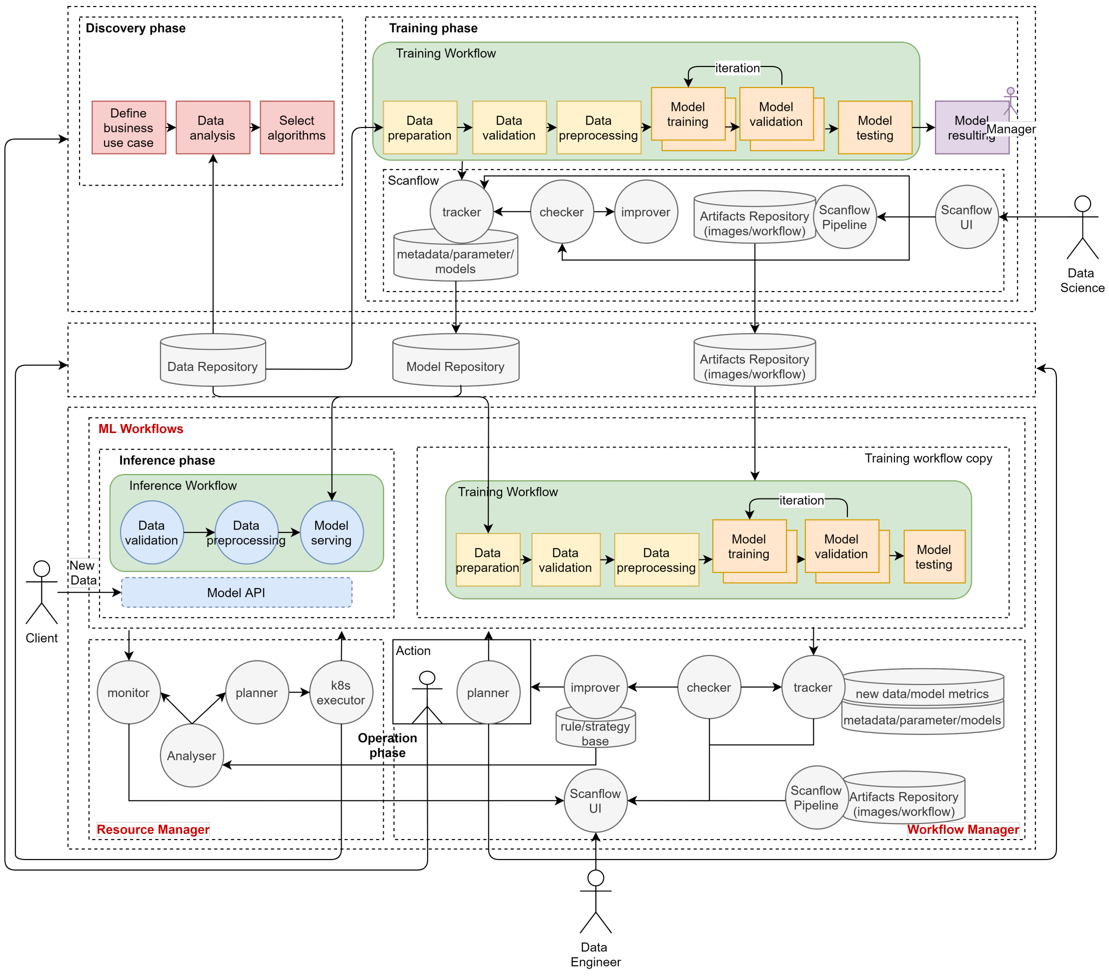


Figure 5.7: Scanflow-K8s for MLOps.

scale the service to achieve the reliability and efficiency of the model. The definition of each step consists of setting the images, requirements, python scripts, and parameters. This definition is set just once and the behavior of each step can be changed by its parameters. In a production system, this notebook should be run once in order to start the network, tracker, executors, and agents as containers. Then, these containers can be executed or reached on demand by using Scanflow API (*e.g.,* call the online predictor service or execute the inference batch executor).

Scanflow-K8s as a shared tool between teams, can help different teams working under the same concept in order to communicate and share the data, models, and artifacts. Also, Scanflow-K8s deals with the hard point within all the stages, thus can help teams fast and easily develop, build, deploy, and auto-manage their workflows. Our MNIST case study is organized in this way. In Section 5.5 we show how different teams use Scanflow-K8s.

5.5 Case Study and Experimental Analysis

This section presents case studies and conducts experiments on Scanflow-K8s to illustrate the features of the agents and evaluate the feasibility and effectiveness of our agent-based approach for autonomic management of ML workflows.

5.5.1 Experimental Setup

Hardware: Our experiments are executed on a ten-node K8s cluster. Each host consists of 2 x Intel 2697v4 CPUs (18 cores each, hyperthreading enabled), 256 GB RAM, 60 TB GPFS file system, and 1-Gigabit Ethernet network.

Software: For all the hosts, we use CentOS release 7.7.1908 with host kernel 3.10.0-1062.el7.x86_64. The Scanflow-K8s platform[10] is built based on Kubernetes v1.19.16 (with Docker 19.03.11, Etcd 3.4.9, Flannel 0.15.0, CNI 0.8.6, and CoreDNS 1.7.0). Its corresponding toolkits (as described in Section 5.2.3) are Istio v1.11.4, Prometheus v14.3.0, Volcano v1.2.0, Keda v2.4.0, Argo Workflows v3.0.0-rc3, and Seldon Core v1.11.2. Additionally, we use Scanflow v0.1.1 with built-in agents for drift detection, which works with MLflow v1.14.1 integrated with a relational database (*e.g.,* PostgreSQL v13.4) for backend entity storage, and an S3 bucket (*e.g.,* Minio Operator v8.0.10) for artifact storage. For the Docker containers used as steps of the ML workflows, Scanflow provides a base executor image using continuumio/miniconda3 and a base service image using python:3.7-slim.

Datasets and Benchmarks: For the first experiment, we use MNIST[11] (60,000 28×28 pixel grayscale images of handwritten digits from 0 to 9) dataset for training a baseline model, and MNIST-C[12] (handwritten digit database with 15 corruptions: corrupted version of MNIST) dataset as new input samples to make predictions.

For the second experiment, we use MLPerf Inference benchmark[13] (details show in section 2.1.3) to test batch and online ML inference for image classification, in particular, we use ResNet50 tensorflow model for the ImageNet2012 validation dataset (50,000 images of objects from 1,000 classifications).

[10] `https://github.com/bsc-scanflow/scanflow`
[11] `http://yann.lecun.com/exdb/mnist`
[12] `https://github.com/google-research/mnist-c`
[13] `https://github.com/mlcommons/inference`

To support batch inference in MLPerf, we extended it with the tf2 backend, which supports tensorflow saved_model format, and we packaged both the model and the serving framework in a Docker image, along with a start script to configure MLPerf when launching the container. To support online inference in MLPerf, we extended it with a Seldon backend, so that MLPerf queries can be generated as RESTful invocations and sent to the model serving services. These extensions are available at Github[14].

MLPerf benchmark supports different realistic end-user scenarios through its LoadGen tool. We use the *Offline* scenario, which represents applications where all data is immediately available and latency is unconstrained, to test the throughput (*i.e.,* samples/s) of batch inference workflows, and the *Server* scenario with multiple concurrent LoadGen clients sending queries according to a Poisson distribution to test the throughput (*i.e.,* queries/s) subject to a latency bound (*i.e.,* 6 ms) of online inference workflows.

5.5.2 MNIST Classification

In this experiment, we show how the various teams will use Scanflow-K8s in the different phases to build and deploy their workflows, as well as the effectiveness of agents that help to manage and supervise the workflows at the application layer while running in production (*i.e.,* detect and handle drift anomalies). **The complete use-case is available at Github[15].**

(1) · Various teams build and deploy workflows

- Training Phase (see Figure 5.8): The Data Science team is responsible for training the ML model to classify MNIST images. Scanflow-K8s supports the definition, building, and execution of batch ML workflows, and runs the various steps of the workflow (*i.e.,* the executors) on Kubernetes by using Argo. Scanflow-K8s allows the modeling step of this workflow to train with different algorithms or with different hyperparameter tuning. Then, the team could select the best model based on the accuracy.

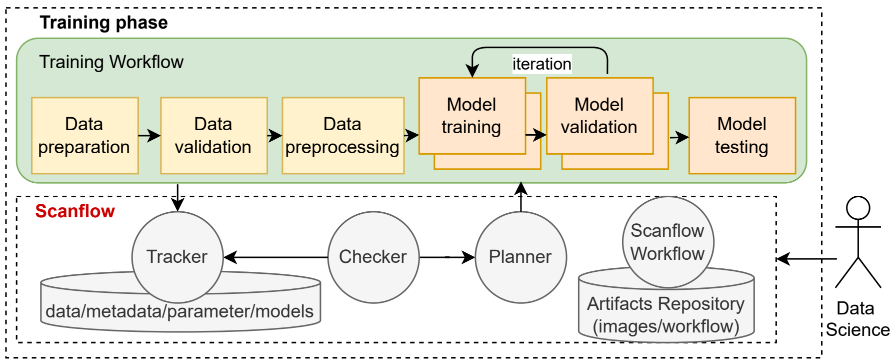


Figure 5.8: Data Science team works at training phase.

- Inference Phase (see Figure 5.9): After the training, the model is stored in the registry provided by MLflow and is ready to be used in production. The Data

[14]https://github.com/peiniliu/inference/tree/scanflow
[15]https://github.com/bsc-scanflow/scanflow/tree/main/tutorials/mnist

Engineer team should build an inference workflow, so that the trained model can be used to make batch predictions, or deployed as a serving service to allow users to ask for predictions online.

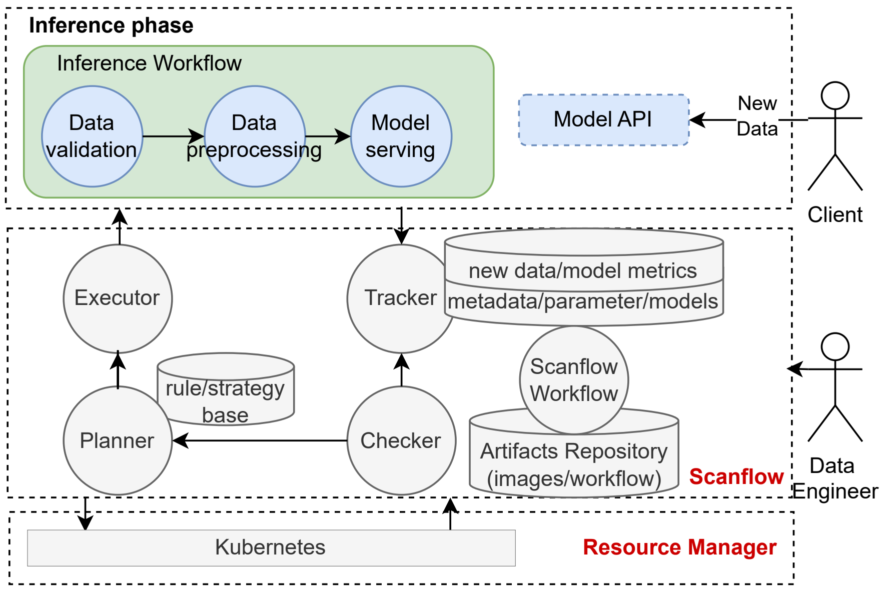


Figure 5.9: Data Engineer team works at inference phase.

② · Agents implementation

Scanflow agents are responsible for application-layer automation. The four internal supported templates of agents are namely tracker-agent, checker-agent, planner-agent, and executor-agent. The Data Engineer team can provide custom functions to enhance the capabilities of each agent. As a proof-of-concept, Algorithm 1 outlines the interaction and collaboration of built-in Scanflow agents which feature a non-trivial drift anomaly detector that autonomically deals with out-of-distribution samples in the data and improves a target accuracy estimator.

This section evaluates agents which feature a non-trivial data drift detector workflow built from the implementation of the components presented in our previous paper [20]. Checker-agent detects out-of-distribution samples by means of a convolutional deep autoencoder and selects the critical points within these data, which are labeled based on human intervention. Planner-agent leverages transfer learning from the original training workflow to retrain the model after adding the labeled picked critical points to the training data. The autonomic strategies of those agents to manage drift are described in detail in Table 5.5. Scanflow-K8s provides an agent framework to help Data Engineers fast build and deploy their agents. For instance, Listing 5.4 - Listing 5.6 show how we implement an autonomic strategy (Algorithm 1 lines 1-2 and Table 5.5 tracker agent) within an internal tracker agent through Scanflow-K8s platform. Data Engineers could define custom strategies in the same manner.

Algorithm 1 Agent-based *model debugging*

Input: Tracker-agent, Checker-agent, Planner-agent, Executor-agent, $newdata$: new predictions samples, m: current model, q: current model accuracy
Output: m': improved current model, q': improved model accuracy

```
1:  while interval = 1 h, size($newdata$) $\geq$ 1000 do
2:     Tracker-agent($newdata$) call Checker-agent;
3:     $anomalydata, pickeddata \leftarrow$ Checker-agent: Detect($newdata$);
4:     while interval = 1 h, size($pickeddata$) $\geq$ 100 do
5:        Checker-agent($pickeddata$) call Planner-agent;
6:        $m', q' \leftarrow$ Planner-agent: Retrain($pickeddata$);
7:        if $q' > q$ then
8:           Planner-agent($m', q'$) call Executor-agent;
9:           $m$ replaced by $m' \leftarrow$ Executor-agent($m'$)
10:       end if
11:    end while
12: end while
```

```python
from .custom_rules import *
from .custom_actuators import *
from typing import List
from datetime import datetime
import time
from functools import reduce
from scanflow.agent.sensors.sensor import sensor

def tock():
    print('Tock! The time is: %s' %  time.strftime("'%Y-%m-%d %H:%M:%S'")
   )

#example 1: count number of predictions
@sensor(nodes=["predictor"], filter_string="tags.mlflow.runName='
   predictor-batch' and metrics.n_predictions > 0")
async def count_number_of_predictions(runs: List[mlflow.entities.Run],
   args, kwargs):

    n_predictions = list(map(lambda run: run.data.metrics['n_predictions'
   ], runs))

    number_of_predictions = reduce(lambda x,y : x+y, n_predictions)

    if number_of_predictions_threshold(number_of_predictions):
        await call_analyze_check_predictions(run_ids = list(map(lambda run
   : run.info.run_id, runs)))

    return number_of_predictions
```

Listing 5.4: Tracker Internal Events.

```python
def number_of_predictions_threshold(number_of_predictions: int):
    return number_of_predictions > 999
```

Listing 5.5: Tracker Internal Constraints.

```python
from scanflow.agent.actuators.actuator import actuator
from typing import List
import mlflow

@actuator(path="/sensors/analyze_check_predictions", depender="checker")
def call_analyze_check_predictions(args, kwargs):
    return args, kwargs
```

Listing 5.6: Tracker Internal Actions.

Table 5.5: Agents autonomic management strategy

Agent	***Strategies***

Tracker-agent	Strategy: *count_number_of_predictions*
	WHEN $IntervalTrigger(1h, count\_number\_of\_predictions)$
	IF $number\_of\_predictions \geq 1000$
	THEN $Call(Checker\text{-}agent : check\_predictions(newdata))$

Checker-agent	Strategy: *check_predictions*
	WHEN $CallReceive(check\_predictions(newdata))$
	IF *successful_call*
	THEN $runWorkflow(Detector\text{-}workflow, newdata)$

Planner-agent	Strategy: *retrain_model*
	WHEN $IntervalTrigger(1h, count\_number\_of\_pickeddata)$
	IF $number\_of\_pickeddata \geq 100$
	THEN $runWorkflow(Training\text{-}workflow(production\_model, retrain = True), pickeddata)$
	Strategy: *update_model*
	WHEN $IntervalTrigger(1h, modelaccuracy)$
	IF $newmodelaccuracy > currentmodelaccuracy$
	THEN $Call(Executor\text{-}agent : change\_model(version))$

Executor-agent	Strategy: *change_model_transition*
	WHEN $CallReceive(change\_model(version))$
	IF *successful_call*
	THEN $updateWorkflow(modelversion, modeltransition)$

③ · Application-level autonomy results

Figure 5.10 presents how a model is autonomously improved by multiple agents in a single interval (model V_1-V_2 time interval). At 60 min, tracker-agent sums up the number of predictions during the last one hour (*i.e.,* interval between blue dashed lines: 0-60min). As there are 1000 predictions, checker-agent is triggered to detect the anomalous data (300 anomalous samples are identified) and pick enough new critical data to be appended to the training dataset. As there are 100 new critical samples, planner-agent is triggered to retrain the model and generate a new version. Only those models trained that achieve better accuracy will be iteratively upgraded by executor-agent to be used in production. Figure 5.11 shows such roadmap of MNIST model upgrades in production. Model V_1 is a baseline model trained by the Data Science team at the training phase with in total

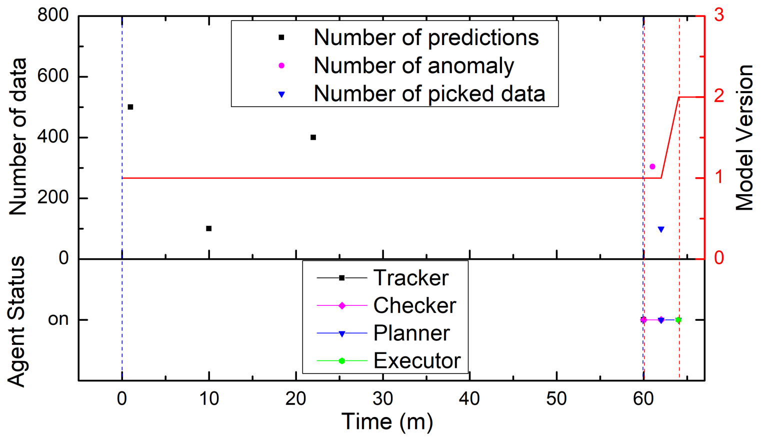


Figure 5.10: Agent-based model debugging in the presence of data drift.

60000 samples and gaining 90% accuracy. The agents monitor predictions over each 1-hour interval (between blue dashed lines) and trigger anomaly detection (between red dashed lines), which might generate a new version of the model for each interval. From those upgraded models, over time only V_2, V_3, and V_7 have been used for predictions in production because they provided better accuracy than the former ones (*e.g.,* V_2: 91%, V_3: 92%, and V_7: 93%). This demonstrates that Scanflow agents can provide autonomy at the application level to help ML workflows to maintain the model accuracy when facing constantly evolving data profiles.

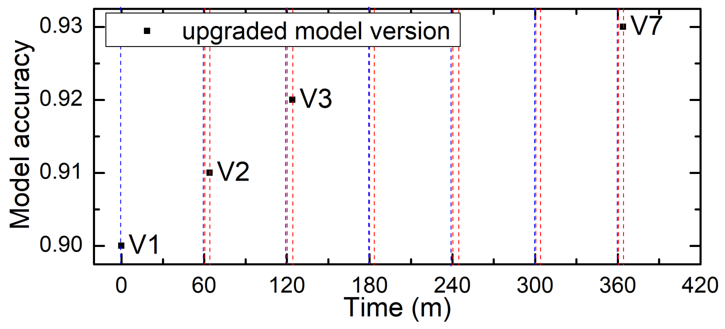


Figure 5.11: Road map of MNIST model upgrades in production.

5.5.3 MLPerf Inference Benchmark

In this experiment, we show how Scanflow-K8s can deal with both context changes and non-functional requirements by taking advantage of the resource manager and also the collaboration between application and infrastructure layers. **This use case is also available at Github**[16]**.**

[16]https://github.com/bsc-scanflow/scanflow/tree/main/tutorials/mlperf

① · Automation at the infrastructure layer

Automation at the infrastructure layer allows taking advantage of the resource management capabilities of the orchestrator to improve the reliability, scalability, and load balancing of workflows. The infrastructure layer provides simple strategies to deal with some system contexts such as self-healing, auto-scaling based on observed system metrics such as CPU utilization, and load-balancing in a round-robin option [88]. However, as they use low-level system information, these strategies are less expressive and more difficult to configure for the end-user, as demonstrated in the next section.

② · Multi-layered Control for Autonomic ML workflows

This section shows the benefit of considering application-provided knowledge to perform resource management actions.

First, we compare infrastructure- vs. application-level auto-scaling by using 100 LoadGen users asking for predictions in the *Server* scenario while expecting a given service QoS (*e.g.,* average queries/s per replica < 20). In Figure 5.12, the data-engineer uses different infrastructure-related settings to define the auto-scaling threshold (*i.e.,* setting the target CPU utilization to 5, 8, or 10 CPUs). The workflow is rapidly scaled up (*i.e.,* number of replicas is increased) at the beginning when setting CPU utilization threshold to 5, hence the system wastes many resources to fulfill the throughput requirement. The workflow is never scaled when setting CPU utilization threshold to 10, thus does not mostly satisfy the throughput requirement. Setting CPU threshold to 8 mitigates the problems of the other two settings, but it is still not matching exactly the QoS requirement. This shows how hard is for the data-engineer to find the optimal auto-scaling settings when using only infrastructure-related metrics. Figure 5.13 shows agent-tuned auto-scaling according to an application-level non-functional QoS requirement provided by the end-user. The planner-agent can autonomically replace the data-engineer to tune the auto-scaling threshold of Keda to meet the requirement. The workflow is scaled up when the real-time throughput goes over the threshold, and scaled down when facing a low load. That is to say, having the application-layer knowledge allows the agents to manage resources wisely by matching the threshold with the QoS requirement in the service level agreement.

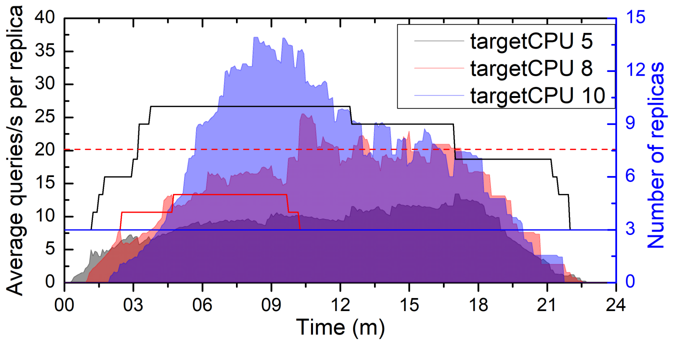


Figure 5.12: Auto-scaling driven by CPU utilization metric.

At this point, we evaluate the agent-tuned anti-affinity for batch workflows, which allows to constrain which nodes they are not eligible to be scheduled based on the pods that are already running on the nodes. We use 50 LoadGen users in the *Server* scenario

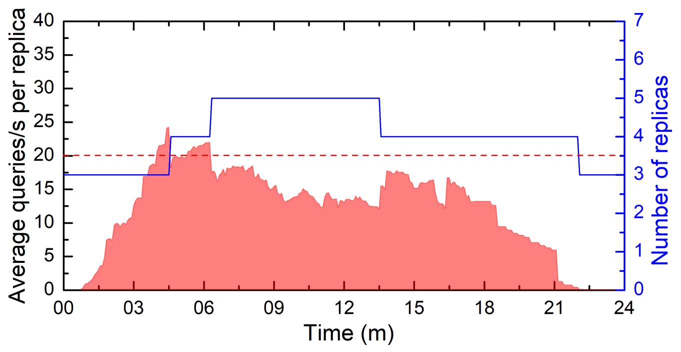


Figure 5.13: Agent-tuned auto-scaling driven by application-level metric.

to stress out an online inference service, while in the meantime another LoadGen user asks for a batch prediction by means of the *Offline* scenario. In the baseline configuration, the batch and online inference workflows are colocated in the same node; while in the anti-affinity configuration, the planner-agent sets anti-affinity of the batch vs. the online workflow, so that they are allocated separately. Figure 5.14 shows the benefit on the performance of both workflows when agents define their anti-affinity, because each workflow can use the spare resources in its allocated node, which would be otherwise used by the colocated workflow if they are executed together, as shown in the baseline.

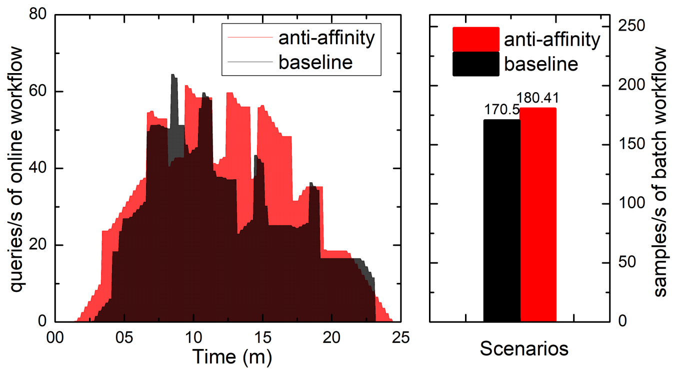


Figure 5.14: Agent-tuned anti-affinity.

Finally, we demonstrate how Scanflow-K8s can deal with workflow internal faults by means of replica fail-over driven by application-level information. In particular, if the inference service is not available and it cannot be recovered by restarting the service instances at the infrastructure layer, a backup service deployed at the initiative of the Data Engineer team can take over and Scanflow-K8s redirects all the traffic from the original inference service to the backup service to maintain the availability. We show the queries' distribution between these two services in Figure 5.15. We have started 200 LoadGen users in the *Server* scenario so that replicas of the original inference service start to fail the readiness health-check due to the high load. When the planner-agent detects that the online-inference service is not available (*i.e.*, its number of ready replicas is 0), it dynamically redirects the query traffic from the unavailable service to the backup service. This is possible thanks to the application knowledge that both services are equivalent, since from the infrastructure perspective they are different services.

The above experiments exemplify how the agents can leverage application-layer knowl-

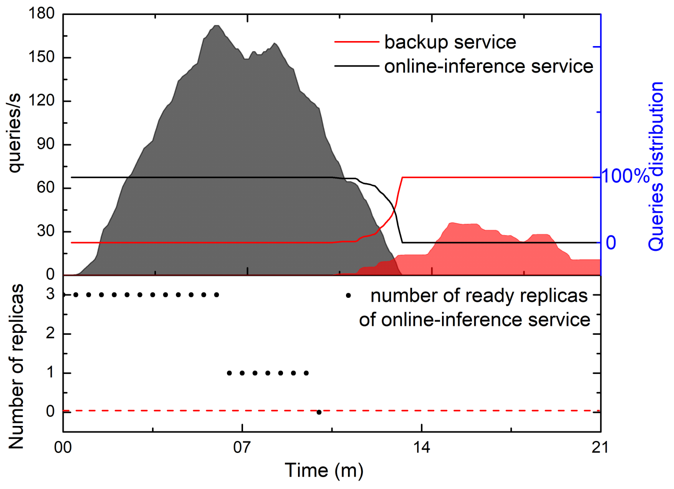


Figure 5.15: Agent-tuned service fail-over and traffic redirection.

edge to enhance resource management actions. In the first one, the agent used arbitrary application-level metrics to configure auto-scaling according to QoS requirements. In the second one, the agent tuned the container-level resource and affinity configuration to optimize performance according to workflow type and resource availability. In the last one, the agent dealt with service unavailability by redirecting the traffic to a backup service defined at the application level. Application-layer knowledge is currently provided by the end-user/data-engineer, but the agent strategies could be enhanced to gather knowledge from other sources (*e.g.,* other models, expert knowledge base).

5.6 Conclusion and Future Work

This chapter presented Scanflow-K8s, an agent-based framework that enables autonomic management and supervision of the end-to-end life-cycle of ML workflows at Kubernetes clusters. We evaluated two use cases, although we engineered the framework so that it can be easily adapted to different ML workloads and more complex adaptation scenarios. First, we used a MNIST project to show how different teams could leverage Scanflow-K8s to manage ML workflows at different phases and how its agents collaborate to debug a drift anomaly problem and upgrade a new model. Second, we used ImageNet2012 classification from MLPerf benchmark for batch and online inference scenarios to show how agents take actions to keep the performance and availability of workflows in this multi-layer controlled autonomic architecture. We provided some template agents to be used in these use cases.

In future work, we plan to implement more generic template strategies and user interfaces so that developers could easily bring their knowledge or the insights learned from other models to the agents. We will also develop more complex (and more dynamic) adaptation policies both at the application and the infrastructure layers, and the needed enhancements in the framework to enforce them at scale (management of conflicts among multiple strategies, agent throughput under high load, etc.).

Chapter 6

Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters

This chapter is based on a conference publication:

[**1**] Peini Liu and Jordi Guitart, "Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters", *The 2022 High Performance Computing and Communications (HPCC-2022)*, December 2022, Chengdu, China, DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00068. (CORE RANK B)

[**2**] May 29, 2021 - Software Release on the Github repository: "Scanflow-Kubernetes: An MLOps Platform". Available at: `https://github.com/bsc-scanflow/scanflow`.

- M3: (27/05/2022) Release mpi branch Scanflow(MPI)-Kubernetes enhanced MPI applications support, Available at: `https://github.com/bsc-scanflow/scanflow/tree/mpi`.

This chapter proposes fine-grained scheduling for containerized HPC workloads in Kubernetes clusters. A brief introduction is presented in Section 6.1. Section 6.2 presents the architecture of Scanflow(MPI)-Kubernetes platform. Detailed algorithms of fine-grained scheduling in different layers are proposed in Section 6.3. Section 6.4 evaluates the performance of our proposed fine-grained scheduling policies for HPC workloads through typical HPC MPI benchmarks. Finally, the conclusions and future work are presented in Section 6.5.

6.1 Introduction

Modern computing infrastructure is evolving at a fast pace to Cloud computing services. Containerization, as a fundamental technology for Cloud computing, allows efficient utilization and easy maintenance of the infrastructure. So far, this attractive paradigm has also had an impact on High Performance Computing (HPC) [18][9].

Previous works have demonstrated the possibility of enabling HPC workloads on Cloud infrastructure using containers [15], and have discussed some best practices for HPC workloads on the Cloud [188][52]. The deployment of containerized HPC workloads in the Cloud is done by container orchestrators, which have the capability to launch and manage containers and their full life cycles, and leverage resource availability and the user specifications to decide the placement of containers. Several orchestrators are available nowadays such as Docker Swarm [33], Mesos [59], and Kubernetes [89]. Kubernetes has been widely adopted in commercial production systems, such as Google Kubernetes Engine [47], and Azure Kubernetes Service [12], and provides a wide and active toolkit ecosystem.

Currently, Kubernetes is not optimized for the management of HPC applications. It is mainly used to support the autonomous management of loosely-coupled long-lived online microservices, enabling their self-healing and auto-scaling. Although it also includes some support for short-lived batch jobs, the tuning of their specification, scheduling, and management must rely on other algorithms and tools. For example, Kubeflow MPI operator [83] provides a better specification for MPI applications and Volcano [186] provides some plugins to enable optimized scheduling for jobs. As the HPC community has important performance considerations on its workloads, developing new deployment schemes for different types of HPC workloads that improve their performance is needed.

Our previous systematical performance analyses in chapter 3 have demonstrated through standalone executions that some types of containerized HPC applications achieve better performance when exploiting multi-container deployments which partition the processes that belong to each application into multiple containers in each node and when constraining each of those containers to a single NUMA (Non-Uniform Memory Access) domain or pinning them to specific processors [99][101]. However, these deployment schemes have not yet been integrated with multi-programmed environments for HPC workloads by current Cloud orchestrators.

In this chapter, we look for fine-grained scheduling policies for allocating containerized HPC workloads through Kubernetes rather than a traditional batch system. The goal is to introduce our optimized management framework to inspire HPC community developers and operators on how to deploy their workloads in a fine-grained way to gain performance improvement and leverage containerization and orchestration technologies. Our main contributions are:

- We present a two-layer scheduling architecture. In the application layer, an agent decides the constructs/granularity of the tasks within the HPC workload based on the characteristics of the applications. In the infrastructure layer, an MPI-aware plugin and task-group scheduling scheme are enabled within a containerized platform scheduler. The MPI-aware plugin decides each task-container mapping and the resource requirements/limits of each container. The task-group scheduling scheme is used to allocate containers to available nodes.

- We develop policies for the granularity-aware agent in the application layer, and the MPI-aware plugin and the container-based task-group scheduling scheme in the infrastructure layer.

- We establish a real platform (so-called Scanflow(MPI)-Kubernetes), implement the algorithms in both application and infrastructure layers, and evaluate the containerized HPC workloads deployments with our fine-grained scheduling scheme.

6.2 System Architecture

Our fine-grained scheduling approach for containerized HPC workloads is built over the existing Scanflow-Kubernetes platform [97][98]. It is implemented both within a Scanflow(MPI) extension package in the application layer (see in Scanflow-Kubernetes github repository[1]) and an enhanced Volcano scheduler/controller manager in the infrastructure layer (see in Volcano github repository[2]). The whole architecture of this platform is depicted in Figure 6.1.

Target System: The yellow area in the figure shows the target system focusing on the HPC workloads. From a static design perspective, HPC workloads are defined as

[1]https://github.com/bsc-scanflow/scanflow/tree/mpi
[2]https://github.com/peiniliu/volcano/tree/peini

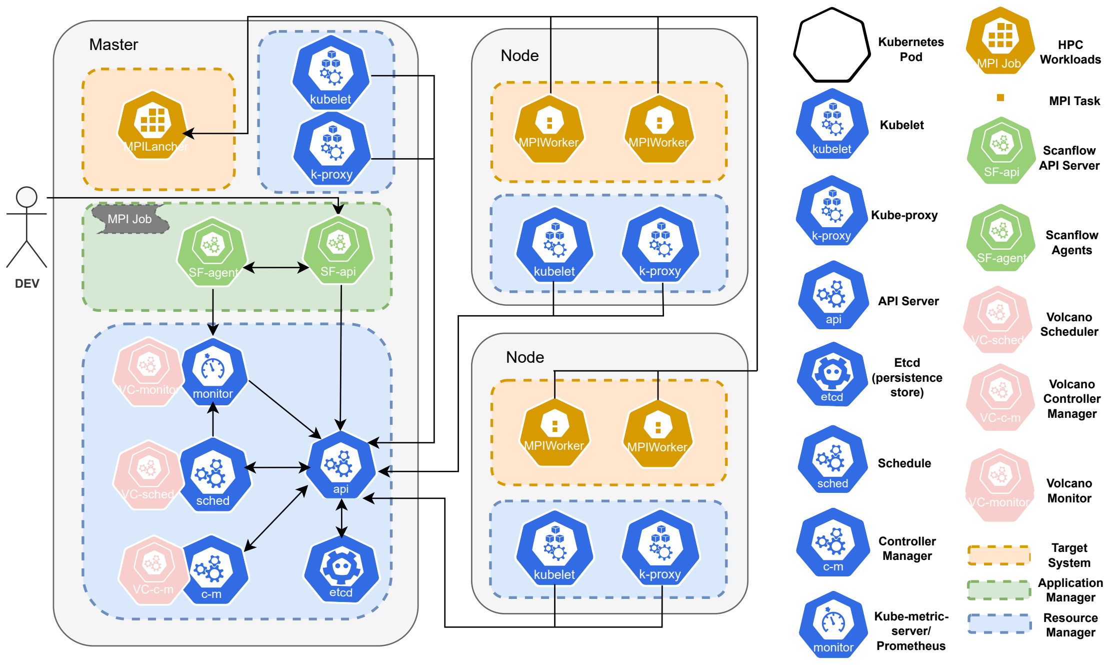


Figure 6.1: Scanflow(MPI)-Kubernetes: A practical platform for managing HPC workloads.

distributed jobs. Typically, an MPI job in the Cloud is composed of a launcher and one or several workers, and all the MPI processes of the job are executed within the workers [83]. However, following the idea of using containerized instances to decouple the processes and considering the potential benefits of multi-container deployments for HPC workloads [99][101], each worker can be split into several finer-grained workers which hold part of processes and are executed in parallel on each node. From a dynamic implementation perspective, the launcher and workers of a job are conducted as containerized instances (*i.e.,* Kubernetes Pods) executing together in the Cloud. All the Pods belonging to the job run once for each time the job is submitted.

Application Manager: Application manager (*i.e.,* Scanflow) is used as a controller of the application layer, as shown in the green area of Figure 6.1. To work with HPC workloads, we implemented a Scanflow(MPI) extension, which allows the users to use the Scanflow-client Python library to easily define and build HPC workloads locally and submit MPI jobs to Scanflow-server to be deployed. This server can connect with Scanflow-agents to calculate proper MPI job granularity (number of workers and nodes to be used) and also submit jobs to Kubernetes Control Plane to run them in a Kubernetes cluster. We also added support for HPC workloads in Scanflow through a granularity-aware planner agent, which can decide the proper granularity for each user-submitted MPI job by considering the provided application profile and the status of the cluster nodes (see Algorithm 2 in Section 6.3).

Resource Manager: The blue area of Figure 6.1 shows the resource manager (*i.e.,* Kubernetes) on the infrastructure layer. Thanks to the Scanflow(MPI) extension described above, our HPC workloads are well-wrapped into containers, thus we can directly use a container orchestrator (*i.e.,* Kubernetes) as resource manager to finely manage the job deployment. We can also take advantage of the wide-range toolkits in the Kubernetes ecosystem, such as Volcano and Prometheus[3].

Kubernetes Control Plane manages the cluster and responds to cluster events. By default, it includes the API Server, Etcd database, and more relevant to this work, Controller Manager and Scheduler. Each type of object has its own Controller Manager to watch its life cycle, for example, Volcano job controller manager watches the job object, and creates the pods to run master/workers to completion. Scheduler watches pods without node assigned and selects the best node for each pod to run on. Node selection has two steps: filtering (to find a set of nodes that are feasible to place the pod) and scoring (to rank the nodes to choose the most suitable placement).

Kubernetes is originally used for managing microservices, thus the default controllers (*i.e.,* Deployment, ReplicaSet) are intended for managing and scaling these applications. Similarly, the policies from the default scheduler are also well-fitted for deploying this type of long-run microservices. However, the usability of the default Job controller and scheduler for deploying HPC workloads is limited. To cope with this, we leverage Volcano into our platform to evolve default jobs into Volcano jobs and change the default scheduler into Volcano scheduler. We also take advantage of the Volcano feature to support additional scheduling plugins to implement an MPI-aware plugin inside the Volcano job controller to configure the *hostfile* and resource request of each worker. Additionally, a task-group scheduling plugin is also implemented inside Volcano scheduler to make scalable and balanced scheduling for fine-grained Volcano jobs (see Section 6.3).

After the global scheduling decided by the Kubernetes Control Plane, pods are stored inside Etcd indicating their assigned node. The next step is to start the pod in the corresponding node through the Kubelet component, which is used for maintaining the pods on nodes (*e.g.,* starting, terminating, reporting). By default, the pods could use requested resources from the whole single node (but not more than their specified limit). However, to do a finer-grain deployment, a CPU/memory management policy should be configured.

HPC workloads can move to different CPUs and increase the context switches if using shared resources, which will degrade the workload performance. As well, related work has shown that CPU/memory affinity could help HPC workloads to gain performance [99][101]. Consequently, we explore different Kubelet settings to allocate exclusive CPUs and/or use NUMA affinity. This paper evaluates two Kubelet settings: (1) default: all pods could use shared resources in a node under the resource limits specification; (2) CPU and memory affinity: sets the `--cpu-manager-policy=static` and the `--topology-manager-policy=best-effort`, so that a pod will be allocated on exclusive CPUs and try best-effort to use CPUs from a single NUMA node.

6.3 Fine-Grained Scheduling

Fine-grained scheduling for containerized HPC workloads is composed of several steps which are shown in Figure 6.2. The notations used are explained in Table 6.1. In the master, the application manager and the resource manager components use their global views to decide the nodes where to allocate the pods belonging to the job, while in each node, the resource manager will decide the resources actually used for each pod that is allocated on this node.

[3]https://prometheus.io/

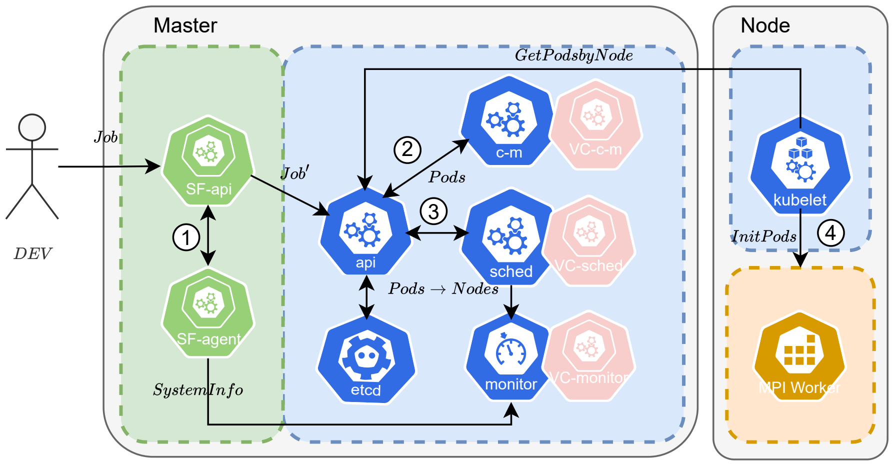


Figure 6.2: Scheduling steps for HPC workloads deployment.

Table 6.1: Notation table.

Notation	Explanation
Job	MPI Job metadata.
N_t	Number of tasks for the Job (fixed).
N_n	Number of nodes for the Job.
N_w	Number of workers for the Job.
N_g	Number of groups of Pods for the Job.
$R(cpu, memory)$	Resource requirements/limits for the Job.
$Pods$	Units to wrap master/workers of the Job.
Pod_w^i	Worker i of the Job.
$Pods_w$	Workers of the Job.
Pod_l	Launcher of the Job.
$Node_j$	Node j in the cluster.
$Nodes$	Nodes in the cluster.
Map($Pod_w^i \rightarrow Node_j$)	Mapping of worker i allocated to a node j.

6.3.1 Application Layer Granularity Selection Algorithm

In the application layer, the developer defines the MPI job (*i.e., Job*), including N_t, which is fixed as it specifies the number of MPI processes this application will start (same as calling 'mpirun -np 16'), and the profile of the application (*e.g.,* network, CPU, memory intensive), which implicitly defines the relevant QoS, and submits it to the Scanflow API Server. Listing 6.1 shows how a user could use Scanflow library to submit their MPI jobs.

```python
import scanflow
from scanflow.client import ScanflowClient
from scanflow.client import ScanflowDeployerClient
client = ScanflowClient()
deployerClient = ScanflowDeployerClient(user_type="local",
                                        deployer="volcano",
                                        scanflow_autoconfig_server_uri = "http://172.30.0.50:49119/sensors",)
# define MPI job
def newWorkflow(i, benchmark, expstr, nTasks, nNodes):
    #mpi workloads
    mpi = client.ScanflowMPIWorkload(name=f"{benchmark}",
                                      mainfile=f"{benchmark}-{expstr}.yaml",
                                      nTasks=nTasks,
                                      nNodes=nNodes,)
    #workflow
    workflow = client.ScanflowWorkflow(type='mpi',
                                        name=f"{benchmark}{i}",
                                        nodes=[mpi],
                                        output_dir = "/home")
    return workflow

#submit Job to Scanflow
async def runWorkflow(i, build_app):
    return await deployerClient.run_workflow(app_name='mpi',
                                  team_name='dataengineer',
                                  workflow = build_app.workflows[i])

build_app = client.build_ScanflowApplication(app = app, trackerPort=46672)
newWorkflow(i, hpccdgemm, "exp1-baseline", 16, 4)
await runWorkflow(i, build_app)
```

Listing 6.1: Scanflow MPI Job Submission.

The Scanflow(MPI) planner agent is responsible for the automatic calculation of other parameters related with the construct/granularity of the *Job* according to a predefined policy set by the admin (see step ① in Figure 6.2), as described in Algorithm 2. In particular, it calculates N_w, N_g, and N_n. For that purpose, the planner agent considers N_t, the application profile, and its resource requirements. If desired, the user can provide a default value for N_w and the agent can get the maximum N_n from Prometheus.

We define two policies, "scale" and "granularity", to determine N_w. In both, each network-intensive application will be packed into a single worker, while the CPU-intensive and the memory-intensive applications will be split into multiple workers, with $N_w = N_n$ in the "scale" policy, and $N_w = N_t$ in the "granularity" policy. If no policy is set, the agent will keep the default N_w specified by the user. Finally, the updated MPI job with granularity will be submitted to the Scanflow API Server, which will transmit *Job* to a Kubernetes cluster through the Kubernetes Control Plane.

Algorithm 2 Granularity Selection (*Planner* agent)

```
Input: $Job$: MPI Job metadata, $SystemInfo$: System information, $Policy$: Granularity policy, $Profile$: Job profile
Output: $Job$: Updated MPI Job metadata with granularity
    {% Agent Sensor: get job specs and system information}
 1: $N_t$, $N_w \leftarrow Job$
 2: $N_n \leftarrow SystemInfo$
    {% Agent Rule: set granularity according to job profile}
 3: if ($Policy$ = "scale") then
 4:     if ($Profile$ = "network") then
 5:         $N_n = 1$, $N_w = 1$, $N_g = 1$
 6:     else if ($Profile$ = "CPU" || "memory") then
 7:         $N_n = min(N_n, N_t)$, $N_w = N_n$, $N_g = N_n$
 8:     end if
 9: else if ($Policy$ = "granularity") then
10:     if ($Profile$ = "network") then
11:         $N_n = 1$, $N_w = 1$, $N_g = 1$
12:     else if ($Profile$ = "CPU" || "memory") then
13:         $N_n = min(N_n, N_t)$, $N_w = N_t$, $N_g = N_n$
14:     end if
15: else
16:     $N_n = 1$, $N_w = N_w$, $N_g = N_n$
17: end if
    {% Agent Actuator: update and submit the job}
18: $Job \leftarrow$ Update($N_n$, $N_w$, $N_g$)
19: Submit($Job$) to Scanflow API Server
```

6.3.2 Infrastructure Layer Task-group Scheduling

In the infrastructure layer, Kubernetes with enhanced Volcano is used to control the lifecycle of the *Job* (see step ② in Figure 6.2) and decide the best nodes to place the *Job* (see step ③ in Figure 6.2). Volcano job controller manager watches the *Job* and creates a *Pod* for each MPI launcher/worker within the job. However, *Job* needs some dynamic configuration while generating the *Pods*. Thus, we enhanced Volcano job controller manager with a plugin implementing Algorithm 3 to make it MPI-aware. This plugin helps *Job* to allocate N_t into N_w in a RoundRobin fashion, decide the $R(cpu, memory)$ for each worker, as well as generate the *hostfile* for all the workers to communicate. After the initialization of the *Job*, all its launcher/workers are wrapped as *Pods* that are registered in Kubernetes API Server and wait for Volcano scheduler to choose the allocated node.

Pods are the smallest deployable entities in Kubernetes, so the scheduler decides their placement individually. However, when enabling granularity, there are various pods that belong to the same job, and we also aim to scale evenly the job into multiple nodes. Therefore, we implemented a task-group plugin inside Volcano (see Algorithm 4). The

Algorithm 3 Dynamic MPI-aware Job Controller

```
Input: $Job$: Job metadata with granularity
Output: $Pods$: Updated pods with resources, $Hostfile$: Hostfile for MPI application
    to allocate tasks
   {% Step 1: get job specification}
1: $Pod_l$, $Pods_w$, $N_t$, $N_w$, $N_n$, $R(cpu/N_t, memory/N_t) \leftarrow Job$
   {% Step 2: allocate tasks into workers in RoundRobin}
2: $nTasksInWorker \leftarrow$ AllocateTasks($N_t$, $N_w$)
   {% Step 3: set up pod resources and the hostfile according to the number of tasks
   allocated }
3: for $i$ in 0 to $N_w - 1$ do
4:     $nTasks \leftarrow$ GetnTasks($nTasksInWorker$, $i$)
5:     $Pod_w^i \leftarrow$ Update($Pod_w^i$, $R(cpu/N_t \cdot nTasks, memory/N_t \cdot nTasks)$)
6:     $Hostfile \leftarrow$ Add(Hostname($Pod_w^i$), slots=$nTasks$)
7: end for
8: $Pods = Pods_w + Pod_l$
9: return $Pods$
```

idea is to group evenly the workers into multiple groups, enabling node affinity for the workers within each group and node anti-affinity among groups. This is done in two steps. First, building multiple groups for every job and allocating worker pods into those groups. Then, filtering for each pod the nodes where it is feasible to schedule it (using Kubernetes default filter), scoring those nodes (using the procedure described in Algorithm 5), and selecting the best one.

Algorithm 4 and Algorithm 5 call some auxiliary functions. For instance, 'sortGroupByResourceRequests' sorts the groups from big to small according to their resource request so that the workers can be evenly added to the groups and each group has similar resource requests; 'WorkerOrderFn' decides the order of the workers taking into account that they can belong to different groups, so it picks up a group and enqueues the workers within the group instead of ordering the workers just by using its id; 'PredicateFn' filters the nodes available to allocate some pods by constraints such as node taints or tolerations; 'NodeOrderFn' in Algorithm 5 calls 'getNodesBoundbyGroup', which returns the node that has been already assigned to the pods in the group, so that when deciding the next pod in the group, the bound node has a higher score.

6.3.3 Node Affinity Settings

In each node, Kubelet takes a set of *Pods* that are provided through the API Server, and starts the containers described in those pods (see step ④ in Figure 6.2). By default, the containers could use requested resources from the whole single node (but not more than the limit), but this paper considers different Kubelet settings to allocate exclusive CPUs and/or use NUMA affinity for containers, as introduced in Section 6.2.

Algorithm 4 Task-Group Scheduling

```
Input: $N_g$: Number of groups, $Pods_w$: Worker pods, $Nodes$: Nodes
Output: $Pods_w$: Worker pods with nodes assigned allocated
    {% Step 1: build and allocate workers into groups}
 1: $groups \leftarrow$ newGroups($N\_g$)
 2: for $i$ in $Pods_w$ do
 3:     $groups \leftarrow$ sortGroupByResourceRequests($groups$)
 4:     $selected\_group = groups[0]$
 5:     AddWorkerToGroup($Pod_w^i$, $selected\_group$)
 6: end for
    {% Step 2: predicate and priority node for worker}
 7: $Pods_w \leftarrow$ WorkerOrderFn($groups$)
 8: for $i$ in $Pods_w$ do
 9:     for $j$ in $Nodes$ do
10:         $pre\_nodes \leftarrow$ PredicateFn($Pod_w^i$, $Node_j$)
11:     end for
12:     for $k$ in $pre\_nodes$ do
13:         $node\_score \leftarrow$ NodeOrderFn($Pod_w^i$, $pre\_nodes_k$)
14:     end for
15:     $best\_node \leftarrow$ getBestNode($max(node\_score)$)
16:     $Pod_w^i \leftarrow$ Update(Map($Pod_w^i$, $best\_node$))
17: end for
18: return $Pods_w$
```

Algorithm 5 NodeOrderFn Node Score Calculation

```
Input: $worker$: Worker, $node$: Node
Output: $score$: score of worker allocated to node
 1: $group \leftarrow$ getGroupByWorker($worker$)
    {% Step 1: base score is the number of bound task in the same group that allocated
    in the node}
 2: $bound\_nodes \leftarrow$ getNodesBoundbyGroup($group$)
 3: for $bound\_node$ in $bound\_nodes$ do
 4:     if $bound\_node = node$ then
 5:         $score++$
 6:     end if
 7: end for
    {% Step 2: count remaining tasks in the same group}
 8: $score = score +$ len($group.worker$)
    {% Step 3: avoid other groups in the node}
 9: for $allocated\_group$ in getGroupsInNode($node$) do
10:     if $allocated\_group \neq group$ then
11:         $score--$
12:     end if
13: end for
14: return $score$
```

6.4 Evaluation

In this section, we evaluate the performance of our proposed fine-grained scheduling policies for containerized HPC workloads through typical HPC MPI benchmarks.

6.4.1 Experimental Setup and Metrics

Hardware: Our experiments are executed on a five-node K8s cluster. Each host consists of 2 x Intel 2697v4 CPUs (18 cores each, hyperthreading disabled), 256 GB RAM, 60 TB GPFS file system, and 1-Gigabit Ethernet network.

Software: All the nodes run Linux CentOS release 7.7.1908 with host kernel 3.10.0-1062.el7.x86_64. The Scanflow(MPI)-Kubernetes platform is built based on Kubernetes v1.19.16 (with Docker 19.03.11, Etcd 3.4.9, Flannel 0.15.0, CNI 0.8.6, and CoreDNS 1.7.0). Its corresponding toolkits (as described in Section 6.2) are Prometheus v14.3.0 and our enhancement of Volcano[2] based on v1.5.0. Additionally, we use Scanflow(MPI)[1] version with built-in planner agent.

Kubernetes Cluster Settings: Our Kubernetes cluster comprises five nodes. We dedicate one node to hold the Control Plane and execute the launcher of MPI applications while the other four nodes are used to run the workers of MPI applications. For each node, we reserve four cores for the system and Kubernetes components, thus 32 cores (16 from each socket) can be used for the allocation of MPI workloads.

As described in Section 6.2, by default Kubelet sets CPU/memory affinity as none. For those experiments that require enabling CPU/memory affinity inside Kubelet, we set it as `--cpu-manager-policy=static` and `--topology-manager-policy=best-effort`.

Scheduler Settings: We use Volcano as default scheduler in the baseline experiments. Volcano is configured by default with the gang plugin enabled, whereas the allocations of all the workers remain the same as Kubernetes default scheduler.

Our fine-grained scheduling policies use two-level scheduling. In the application layer, the granularity selection algorithm is implemented inside the Scanflow planner agent. In the infrastructure layer, we use an enhanced version of Volcano that implements our MPI-aware controller and also features our task-group scheduling.

Benchmark Settings: We use the HPC Challenge benchmark suite[4] and the MiniFE proxy application for unstructured implicit finite element codes[5]. They are built with OpenMPI v4.0.3rc3, and run with 16 MPI processes in exactly-subscribed mode, all of them bound to all the processors allocated to the application (*i.e.,* 16 cores) in all the scenarios.

The specific MPI profile analysis (used to classify MPI applications) can be found in Figure 6.3 and Section 3.2 in Chapter 3. EP-DGEMM and EP-STREAM are MPI throughput applications: the former is CPU intensive and the latter is memory bandwidth intensive. G-RandomRing Bandwidth and G-FFT are MPI communication applications where processes need to communicate (frequently and globally) with each other. For the application MiniFE, we set the problem size as nx=ny=nz=512. As shown in Figure 6.3, it contains some MPI_Allreduce communications (*i.e.,* global reduce) but they can scale without introducing much network latency [60]. Thus the application is categorized as memory- and CPU-intensive.

[4]http://icl.cs.utk.edu/hpcc/
[5]https://github.com/Mantevo/miniFE

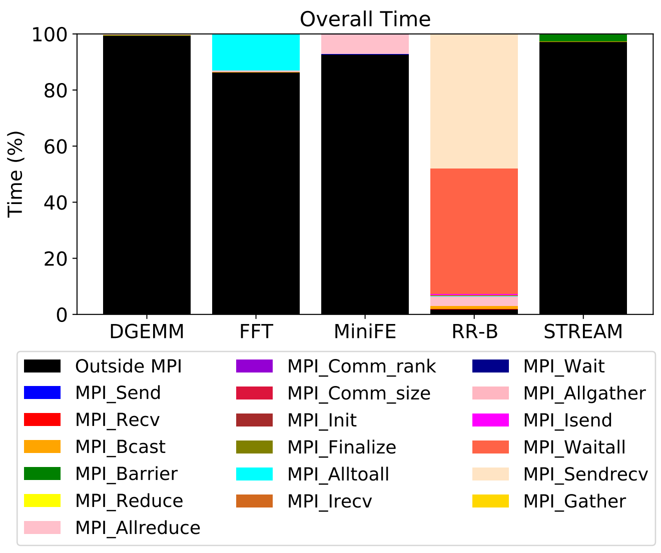


Figure 6.3: Benchmarks MPI profiling analysis.

Scenario Settings: We consider six scenarios (see Table 6.2): *NONE* and *CM* are two baseline scenarios, the former with the default settings of Kubernetes, and the latter with the CPU/memory affinity settings supported by Kubelet. Given the well-known benefit of tuning the CPU/memory affinity for HPC MPI applications, we compare our policies on top of CPU/memory affinity. Scenarios *CM_S* and *CM_G* use two different strategies for agent granularity selection, namely 'scale'(S) and 'granularity'(G), which were described in Algorithm 2. Scenarios *CM_S_TG* and *CM_G_TG* maintain the benefits that the granularity policies apply in the application layer and also show the effectiveness of using our proposed task-group scheduling (TG) (see Algorithms 4-5) in the infrastructure layer. Summarizing, the settings of the six scenarios are as shown in Table 6.2:

Table 6.2: Scenarios settings.

Scenarios	**Kubelet**	**Scanflow**	**Volcano**
NONE	default		default(gang)
CM	cpu/memory affinity		default(gang)
CM_S	cpu/memory affinity	granularity selection 'scale'	default(gang)
CM_G	cpu/memory affinity	granularity selection 'granularity'	default(gang)
CM_S_TG	cpu/memory affinity	granularity selection 'scale'	default(gang)+ task-group scheduling
CM_G_TG	cpu/memory affinity	granularity selection 'granularity'	default(gang)+ task-group scheduling

Metrics: We consider four main metrics in our evaluation:

- **Job Running Time (T_i^r):** the running performance of job i.
- **Job Response Time (T_i):** the total wallclock time from the instant at which

the job i is submitted to the system until it terminates [35]. It is composed of two parts: the time T_i^w that job i is waiting and the time T_i^r that job i is actually running in parallel on multi-processing nodes. Thus, $T_i = T_i^w + T_i^r$.

- **Overall Response Time (T):** the total response time summed from all the jobs. $T = \sum T_i$
- **Makespan ($T_{makespan}$)**: the time required for all jobs to terminate. It is directly linked with utilization and throughput, and each can be derived from the others [36].

6.4.2 Schedule With one Type of MPI Workload

Our previous Chapter 3 showed that EP-DGEMM benchmark can improve its performance thanks to a finer-grain deployment scheme [99]. Thus, firstly we set an experiment with this single type of application, and we submit 10 MPI EP-DGEMM jobs with an arrival interval of 60 seconds.

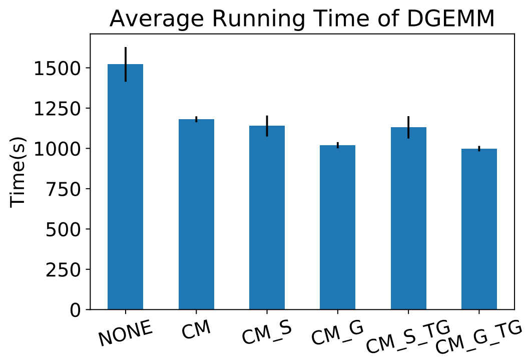


Figure 6.4: Average job running time of 10 EP-DGEMM jobs.

Figure 6.4 shows the average performance of the EP-DGEMM workload in the six scenarios. Scenario *CM* shows better cache utilization (less L3 misses), more local memory accesses, and only minimal remote memory accesses than *NONE* scenario. When enabling the 'scale' and the 'granularity' policies, we partition each application with more number of containers, but less number of processes per container. Those scenarios, namely *CM_S**(*i.e.*, *CM_S* and *CM_S_TG*) and *CM_G**(*i.e.*, *CM_G* and *CM_G_TG*), have considerably less process migrations and context-switches than *NONE* and *CM* baseline scenarios. Moreover, for *CM_G** scenarios, as each container runs a single process, this is essentially a single-level scheduling (*i.e.*, at the cgroup level), which is simpler and allows to exploit processor affinity better, in a similar way to when processes are pinned explicitly, which is an important factor for the performance of CPU-intensive applications.

As shown in Figure 6.5, the improvements in the running time of DGEMM in those scenarios cause also an improvement in the overall response time. In particular, *CM_S** have 5% and 26% improvement and *CM_G** have 15% and 34% improvement, compared to baseline scenarios *CM* and *NONE*, respectively. Note that *TG* incurs no significant benefit for DGEMM because its CPU requirements can be granted in all cases and this does not introduce imbalance problems.

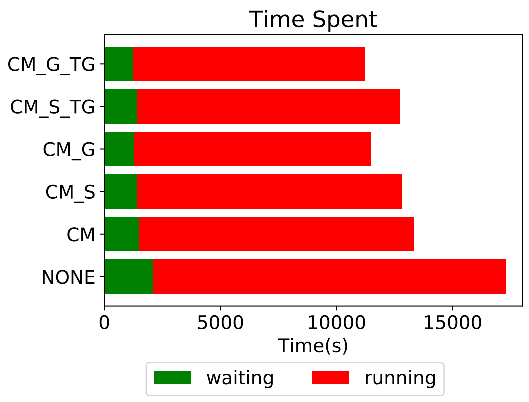


Figure 6.5: Overall response time of scheduling 10 EP-DGEMM jobs.

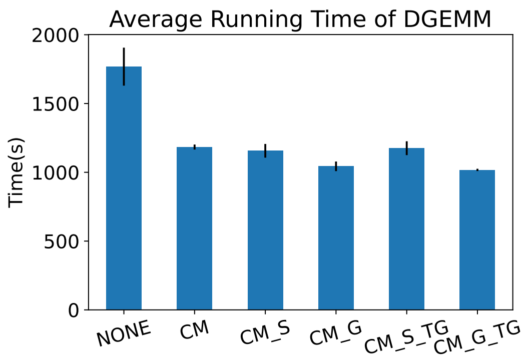
6.4.3 Schedule With Multiple Types of MPI Workloads

In this experiment, we evaluate the effectiveness of our policies to fit different types of workloads. We randomly generate a submission time for 20 MPI workloads within the interval 0 to 1200 seconds. Workloads come from the five benchmarks (*i.e.,* EP-DGEMM, EP-STREAM, G-FFT, G-RandomRing Bandwidth, and MiniFE), and each benchmark will be run 4 times, with a random sequence.

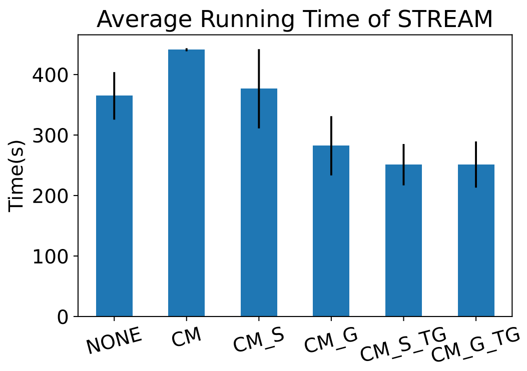
Figure 6.6 shows the average job running time of different types of workloads and the overall response time in the six scenarios. Baseline scenario *NONE* uses shared resources for all the running workloads, thus potentially having computation imbalance as the processes can move among the several CPUs in the node. The randomness of these processes movement can incur a variable performance between different executions of the same type of job, thus impacting the average runtime. Baseline scenario *CM* shows better cache utilization (less L3 misses) and reduces remote memory accesses latency, but introduces more memory contention for memory-intensive applications than *NONE* scenario.

When enabling 'scale' and 'granularity' policies in *CM_S** and *CM_G**, we partition CPU- and memory-intensive applications with more number of containers but less number of processes per container, while the processes within a network-intensive application remain in a single container to avoid the network latency. As shown in Figure 6.6, 'scale' and 'granularity' policies do not have significant effects on the network-intensive applications (*i.e.,* RR-B and FFT), but considerably improve the performance of CPU- and memory-intensive applications regarding the baseline scenarios. Task-group scheduling (TG) has an important impact on memory-intensive benchmarks, for instance, *CM_S_TG* can reduce a 33% the running time of STREAM regarding *CM_S*. This is because by default the scheduler randomly chooses the nodes to deploy the pods within the same job, and some load imbalance could introduce more memory contention and latency. TG uses evenly distribution for jobs to deploy their pods into nodes, thus maximally guaranteeing the balance of MPI applications.

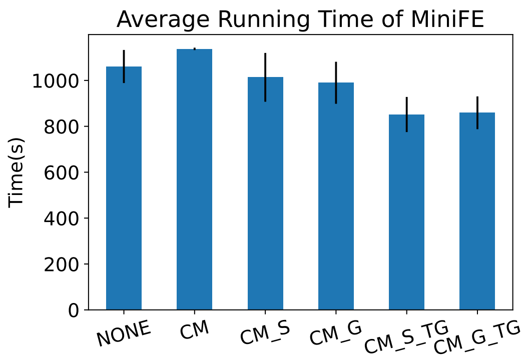
Figure 6.6f shows the overall response time of *CM_S_TG* has 16% and 32% improvement, and *CM_G_TG* has 19% and 35% improvement, both compared to baseline scenarios *CM* and *NONE*, respectively. These come both from the granularity selection, but also from the task-group scheduling, since *CM_S_TG* and *CM_G_TG* have 12% and



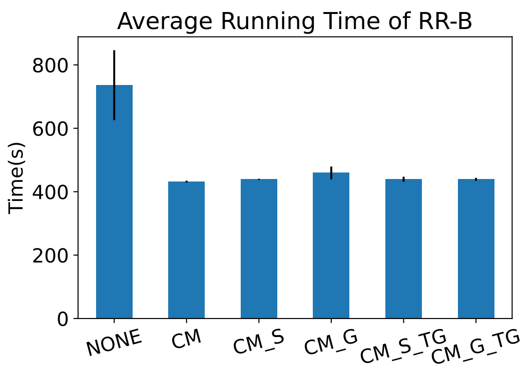
(a) DGEMM Average Running Time



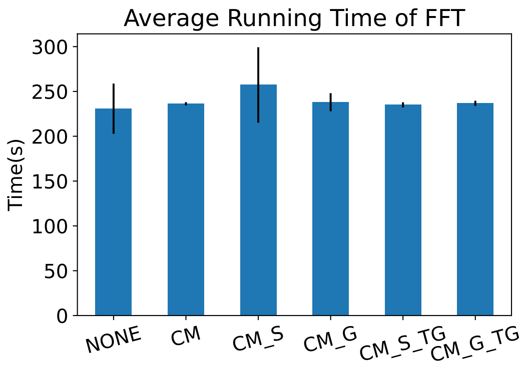
(b) STREAM Average Running Time



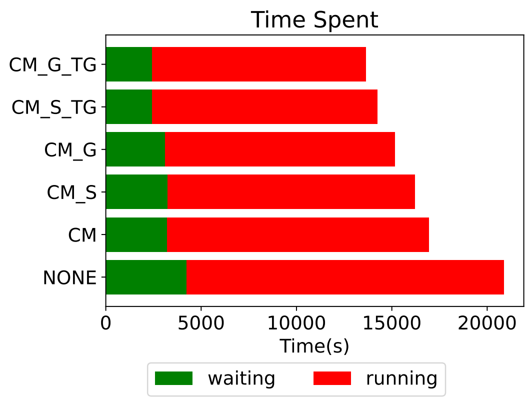
(c) MiniFE Average Running Time



(d) RamdomRing-Bandwidth Average Running Time

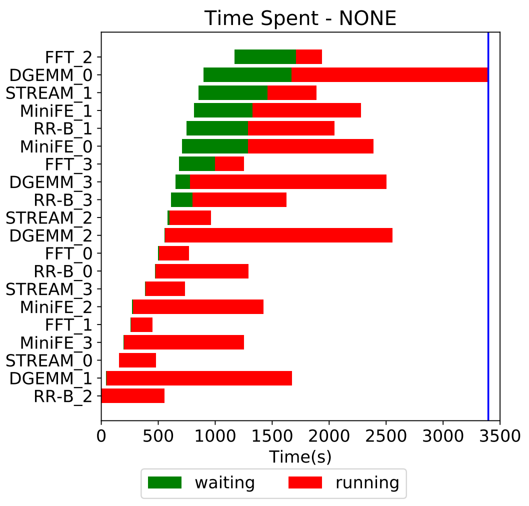


(e) FFT Average Running Time

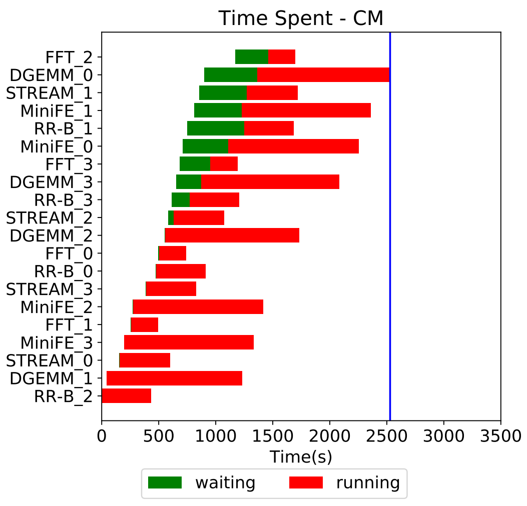


(f) Overall Response Time

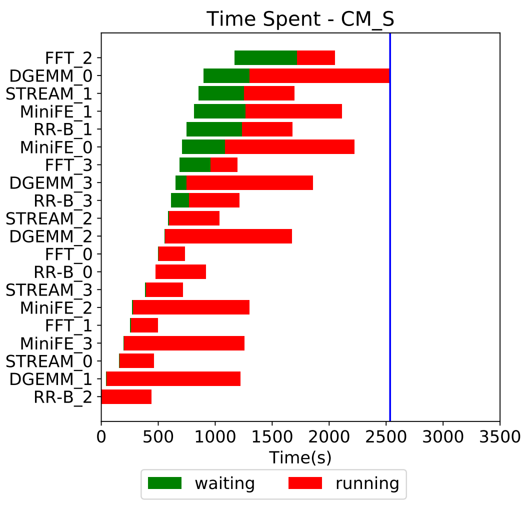
Figure 6.6: Average job running time of each type of workload (Figs. 6.6a-6.6e) and overall response time when scheduling 20 jobs of different types (Fig. 6.6f).



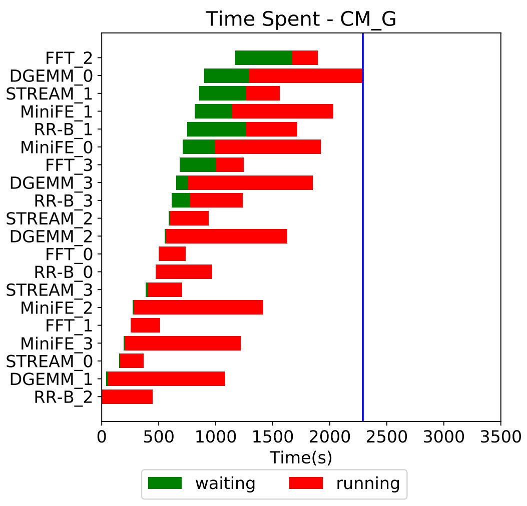
(a) Makespan of *NONE*



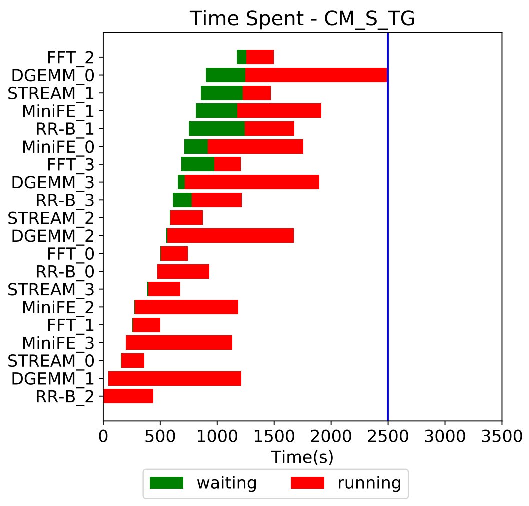
(b) Makespan of *CM*



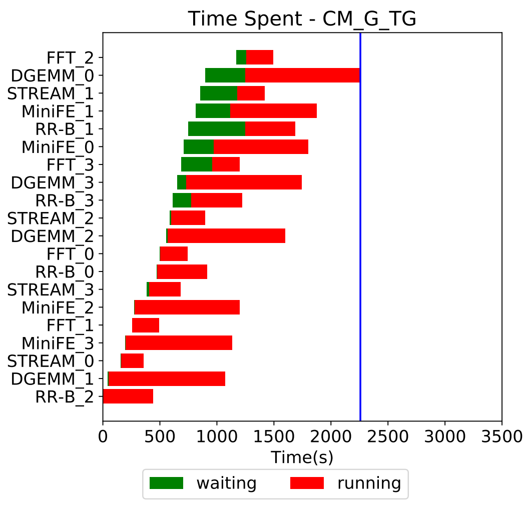
(c) Makespan of *CM_S*



(d) Makespan of *CM_G*



(e) Makespan of *CM_S_TG*



(f) Makespan of *CM_G_TG*

Figure 6.7: Makespan of six scenarios: scheduling 20 jobs of different types.

10% performance improvements with respect to *CM_S* and *CM_G*.

To evaluate the effectiveness of our two-level scheduler for the entire workload, we show the makespan in Figure 6.7, which also presents in detail the scheduling process of each scenario. Scenario *CM_S_TG* has 1% and 26% makespan reduction, whereas scenario *CM_G_TG* has 11% and 34% makespan reduction, both with respect to baseline scenarios *CM* and *NONE*, respectively, which demonstrate how our policies could improve overall system throughput.

6.4.4 Schedule Under Different Frameworks

This experiment compares our approach to schedule MPI workloads with Kubeflow MPI operator [83] and native Volcano [186]. MPI jobs specified by Kubeflow are scheduled by Kubernetes default scheduler. Volcano specifies jobs through its own Job Controller and schedules them using Volcano Scheduler, which features a gang plugin by default. Kubelet for these two scenarios is set with CPU/memory affinity enabled. Other settings are the same as the experiment in section 6.4.3.

As shown in Table 6.3, which displays the makespan for all the evaluated scenarios, *Kubeflow* framework has similar makespan to the *CM* baseline scenario, because both use CPU/memory affinity settings and use the default or default-alike scheduler. *Volcano* framework has an important slowdown on makespan, because it partitions all the workloads, even the network-intensive ones, which incur high latency and contention. Consequently, both frameworks fail to provide better performance than our fine-grained scheduling.

Table 6.3: Makespan comparison.

Scenarios	**Makespan**
Kubeflow	0 days, 00:42:00 (2520 s)
Volcano	1 days, 10:10:55 (123055 s)
CM	0 days, 00:42:09 (2529 s)
CM_S_TG	0 days, 00:41:38 (2498 s)
CM_G_TG	0 days, 00:37:38 (2258 s)

Figure 6.8 shows the job running time of each job. *Kubeflow* has a similar job running time as *CM*, because they do not partition a job into multiple containers, hence CPU- and memory-intensive workloads cannot benefit from multi-container deployments. Contrariwise, *Volcano* allocates a job by default as one process per container, and those containers are randomly submitted to multiple nodes. Consequently, network-intensive workloads face very important performance degradation due to an increasing number of communications. In scenarios *CM_S_TG* and *CM_G_TG*, some of the CPU- and memory-intensive workloads show even better performance than *Volcano* because those scenarios enable the task-group plugin so that the group of fine-grained containers from a same job can be evenly allocated to the nodes.

Figure 6.9 shows the job response time of each job. Our fine-grained scheduling outperforms the rest, in particular, the container allocation in *CM_G_TG* scenario improves (or at least equals) the running time of all the jobs, as well as their waiting time. *Volcano* is the worst case, as network-intensive workloads have an important performance degradation, thus also introducing more waiting time for the following jobs.

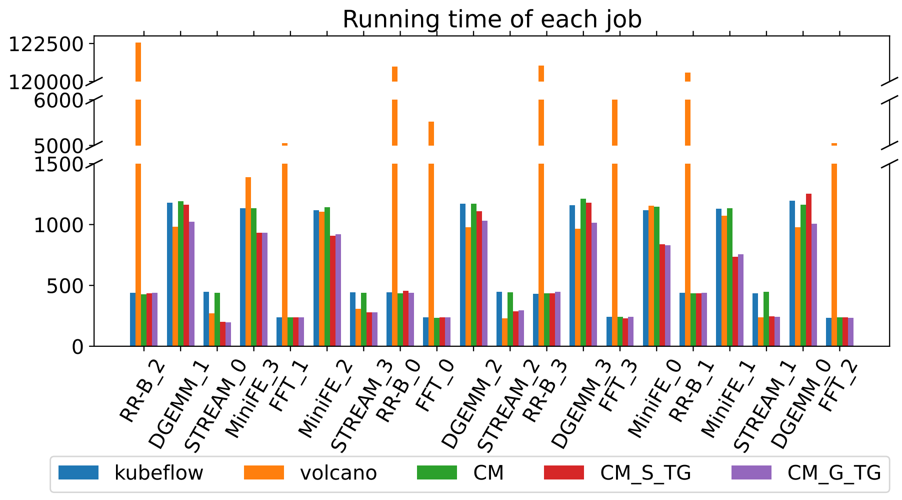


Figure 6.8: Job running time with different frameworks.

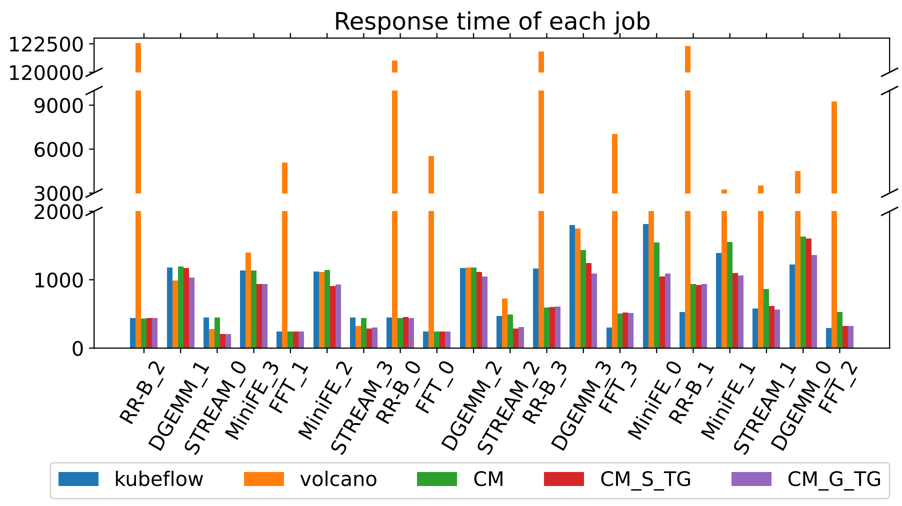


Figure 6.9: Job response time with different frameworks.

6.5 Conclusion and Future Work

This chapter presented fine-grained scheduling policies for allocating containerized HPC workloads in a Kubernetes cluster. We extended the Scanflow-Kubernetes platform to support HPC MPI workloads and improved its two-layer scheduling architecture, by creating new policies in both the application-layer planner agent (*i.e.,* enabling granularity selection), as well as the infrastructure-layer Volcano controller and scheduler (*i.e.,* adding an MPI-aware controller and a task-group scheduling plugin).

Our results show that the proposed fine-grained policies can reduce the response time of HPC workloads up to 35%, as well as improve the makespan up to 34%. Although our benchmarks are small-scaled MPI jobs that fit in a single node, our principles to exploit granularity are also applicable if applications do not fit in a single node: *e.g.,* for network applications, one would probably use coarse-grained granularity within each node to exploit fast shared-memory communication, whereas CPU-bound applications could use fine-grained granularity to exploit affinity. In the future, we will enhance our fine-grained policies for the scheduling of mixed HPC-AI workloads on Kubernetes, and consider other application profiles such as I/O applications. Moreover, we will evaluate them in larger-scale scenarios.

Chapter 7

Conclusions

The convergence of HPC, BD, and ML is being pursued in earnest across the academic and industry in recent years, and it is predictable that it will keep developing in the future. It is important to use a holistic approach that involves a multi-disciplinary team of experts in HPC, BD, and ML, and to use a combination of technologies and approaches. In this thesis, we hypothesized virtualization and containerization technologies can be the basis towards the convergence. In this chapter, our general conclusions, our specific contributions, and the main future research lines are presented, as a closure of the doctoral thesis.

7.1 General Conclusions

In conclusion, the convergence of HPC, BD, and ML is becoming increasingly important in modern computing, and containerization technology can play a critical role in enabling their convergence. This thesis proposed several contributions to leverage containerization technology to converge HPC, BD, and ML applications on containerized infrastructures. Specifically, it focused on supporting the deployment of HPC, BD, and ML applications using containers, analyzing the performance of these applications running on containers with different deployment options, providing an autonomous management platform for containerized HPC, BD, and ML applications, and optimizing container management and scheduling for containerized HPC, BD, and ML applications.

These contributions aimed to address the challenges of deploying and managing these complex applications in a heterogeneous computing environment. Our achievements demonstrated that containerization technologies can support the convergence of HPC and ML applications, not only keeping the well-known advantages of containerization regarding customization, portability, reproducibility, and fault isolation, but providing also performance benefits thanks to fine-grain deployments and resource allocation.

7.2 Contributions

This thesis has presented relevant contributions. The first two contributions focused on a containerization basis (using containers to deploy various applications on different resources, and analyzing the performance of different deployment options). The third contribution towards the platform challenges, established a real two-layer controlled platform to deploy various containerized applications and provide agents for autonomic management. The last contribution provided policies for the platform in multiple layers to address efficient container management.

- **Enabled deployments of HPC, BD, and ML applications using containers**, so that applications could make an efficient use of the containers (through different configurations) to improve their performance.
  - **Chapter 3:** We proposed the multi-container deployments (*i.e.,* partitioning the processes belonging to each application into different containers) for HPC applications, and derived corresponding affinity settings for each container belonging to the deployment scheme. We enabled these configurations both in Docker and Singularity.
  - **Chapter 3:** In addition to the above settings, we also enabled the different network interconnection for multiple containers belonging to an HPC application in an InfiniBand cluster.
  - **Chapter 4-5:** We enabled containerization for ML workflows using Docker, in both ML training stage and ML inference stage. Particularly, we adopted multi-container deployment schemes and affinity settings for online ML inference services.
- **Understood the performance of HPC, BD, and ML applications running on containers**, and gathered knowledge on how to choose the most adequate deployment schemes for containerized applications to achieve the best performance.
  - **Chapter 3:** We contributed a performance analysis of distinct multi-container deployment schemes for HPC workloads comprising i) different containerization technologies, ii) different container granularity, iii) different container processor and memory affinity configurations, iv) different hardware platform settings (*e.g.,* Non-Uniform Memory Access (NUMA), Uniform Memory Access (UMA)), v) different application subscription modes (exactly- or over-subscribed mode). The results showed that a) some types of containerized HPC applications can exploit multi-container deployments which partition the processes that belong to each application into multiple containers in each host in order to achieve better performance; b) some types of HPC applications gain benefits when using containers by constraining them to a single NUMA domain or pining to specific processors.
  - **Chapter 3:** We presented a detailed performance characterization of different containerization technologies (including Docker and Singularity) for HPC workloads on InfiniBand clusters through different dimensions, namely network interconnects (including Ethernet and InfiniBand) and protocols (including TCP/IP and Remote Direct Memory Access (RDMA)), networking modes (including host, MACVLAN, and overlay networking), and processor and memory affinity. The results showed that multi-container deployments and affinity for HPC applications on InfiniBand clusters also show distinct performance benefits while using different networking modes.
  - **Chapter 4:** We presented a performance characterization of multiple deployment schemes for online ML inference services that feature different degrees of container granularity and we set the corresponding distribution of application working threads and resources to each container to serve the model. In addition, we investigated CPU/Memory affinity for each container belonging to an online ML inference service as part of the former deployment schemes. The

performance analysis results demonstrated that multi-container deployments show significant performance improvements for online ML inference services, and finer-grained deployments show better performance because they favour process affinity in a similar way to when threads are pinned explicitly. Also, these deployments fit very well with explicit CPU/memory affinity settings for each container.

- **Established an autonomic management platform for containerized HPC, BD, and ML applications**, which supported accelerating containerized application development and image building, optimized deployment options for efficient deployments, and introduced autonomic computing for continuous application management.
  - **Chapter 5:** We enabled an agent-based approach to leverage autonomic computing. The Scanflow agents focused on maintaining the robustness and satisfying requirements at the application layer. We explained the design of Scanflow multi-agent approach by using triggers, primitives, and strategies.
  - **Chapter 5:** We investigated an architecture with abilities for two-layered management (*i.e.,* application layer and infrastructure layer). Based on the architecture, we established a real platform Scanflow-K8s, a functional agent-based platform that enables autonomic management and online supervision of the end-to-end life-cycle of ML workflows on Kubernetes. Moreover, various teams could use Scanflow-K8s to build and deploy their ML workflows in different phases.
  - **Chapter 6:** We extended the platform in the application layer with a Scanflow(MPI) package, which allowed the users to use the Scanflow-client Python library to easily define and build HPC workloads locally and submit MPI jobs to Scanflow-server to be deployed in a Kubernetes cluster.
- **Optimized container management and scheduling for containerized HPC, BD, and ML applications**, by devising autonomic management and online supervision mechanisms for ML workflows and fine-grained scheduling policies for HPC workloads to adapt to dynamic contexts and provide efficient container orchestration.
  - **Chapter 5:** We conducted experiments on Scanflow-K8s to illustrate the features of the agents and evaluate the feasibility and effectiveness of our agent-based approach for autonomic management of ML workflows. We defined and implemented policies for Scanflow agents to support autonomous management for ML workflows in each different dynamic context.
  - **Chapter 6:** We proposed fine-grained scheduling policies for containerized HPC workloads in Kubernetes clusters, focusing on multi-container deployment according to the application profile, using CPU/memory affinity and the idea of even distribution. We implemented and adopted our scheduling schemes on a Scanflow(MPI)-K8s. We developed policies for the granularity-aware agent in the application layer, and the MPI-aware plugin and the container-based task-group scheduling scheme for the Kubernetes Volcano scheduler in the infrastructure layer.

7.3 Future Work

The convergence of HPC, BD, and ML applications on containerized platforms is already a reality, and it is currently in a clear growing trend that containers are connecting more heterogeneous hardware and different types of applications. The potential future work based on the accomplished achievements of the present thesis is described below:

- Chapter 3:
  - Use insights about the performance of multi-container deployments, especially those regarding the impact of the container granularity and the CPU and memory affinity to derive placement policies when deploying HPC workloads which can get better utilization of the resources while maintaining application performance. A typical investigation has been worked in Chapter 6 where we derived placement policies for containerized HPC workloads using a fine-grain idea and integrated them with a new scheduler in Cloud, such as the Kubernetes native batch scheduling system (*i.e.,* Volcano[1]).
  - Investigate the impact of the performance of containerized HPC workloads deployments using other dedicated hardware/infrastructure, such as GPFS, GPUs, etc.
- Chapter 4:
  - Consider the performance insights in this chapter about the container-level settings (*i.e.,* container granularity and affinity) to derive placement policies integrated within the Kubernetes scheduler/Kubelet agent for the efficient deployment of online multi-model ML inference services in a multi-programmed and multi-tenant Cloud environment.
  - Extend the results, not only considering deployments for the online ML inference services, but also investigating the performance of ML training or inference batch workloads, since they are also important in a ML life-cycle and also resource intensive.
  - Integrate edge devices with Cloud to build a computation continuum and enable containerized ML workloads with this paradigm. For instance, federated learning for distributed model training and inference.
- Chapter 5:
  - Implement more generic template strategies and user interfaces for agents so that developers can easily bring their knowledge or the insights learned from other models to the agents.
  - Develop more complex (and more dynamic) adaptation policies both at the application and the infrastructure layers, and the needed enhancements in the framework to enforce them at scale (management of conflicts among multiple strategies, agent throughput under high load, etc.). Similarly as we have done with the multi-layer fine-grained scheduling policy presented in Chapter 6.
  - Extend the platform with distributed edge resources. The infrastructure resource manager and its policies should also be enhanced to consider different sources of resources and use the resource efficiently and energy-awarely.

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

  - Application of the aforementioned methods to other case studies and different types of applications.

- Chapter 6:
  - Enhance our fine-grained policies for the scheduling of mixed HPC-AI workloads on Kubernetes, and consider other application profiles such as I/O applications.
  - Design and develop a Volcano device plugin to dynamically report information regarding the current status of CPU and memory usage, the scheduler could use this information to do wiser CPU or memory affinity binding. In addition to using an automatic Kubelet agent inside each node to start a container, Volcano device plugin should also help containers to start in a specific cgroup (*i.e.,* CPUSET).

In the next years, the convergence of HPC, BD, and ML applications on the computing continuum will keep growing, and the complexity of mixed container systems will challenge the system stability, reliability, energy-efficiency, as well as the applications' performance, QoS, and robustness. Therefore, further research to improve the state-of-the-art management strategies in both the application layer and the infrastructure layer that ensure robustness and efficiency in large-scale and more complex systems with a higher amount of containerized applications is necessary.

Appendix A

Publications

This section presents a list of the author's journals, conferences, and other publications. Some of them directly correspond to the contributions of this thesis, whereas others are related works that have been done in collaboration with other researchers. Additionally, contributions to the open-source community and some Cloud Native Computing Foundation (CNCF) projects are listed as other contributions.

Publications included in this thesis

Published journal papers

J1 Peini Liu and Jordi Guitart, "Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study", *The Journal of Supercomputing*, vol. 77, no. 6, pp. 6273-6312, June 2021. DOI: 10.1007/s11227-020-03518-1. **JCR Q2**.

J2 Peini Liu and Jordi Guitart, "Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study", *Cluster Computing*, vol. 25, no. 2, pp. 847-868, April 2022. DOI: 10.1007/s10586-021-03460-8. **JCR Q2**.

Published conference papers

C1 Peini Liu, Gusseppe Bravo-Rocca, Jordi Guitart, Ajay Dholakia, David Ellison, and Miroslav Hodak, "Scanflow: an end-to-end agent-based autonomic ML workflow manager for clusters," *In Proceedings of the 22nd International Middleware Conference: Demos and Posters*, December 2021, Virtual Event, Canada. pp. 1-2, DOI: 10.1145/3491086.3492468. **Core Rank A**.

C2 Peini Liu, Gusseppe Bravo-Rocca, Jordi Guitart, Ajay Dholakia, David Ellison, and Miroslav Hodak, "Scanflow-K8s: Agent-based Framework for Autonomic Management and Supervision of ML Workflows in Kubernetes Clusters", *2022 IEEE/ ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2022, Taormina, Italy, pp. 376-385, DOI: 10.1109/CCGrid54584.2022.00047. **Core Rank A**.

C3 Peini Liu and Jordi Guitart, "Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters", *The 2022 High Performance Computing and Communications (HPCC-2022)*, December 2022, Chengdu, China, pp.275-284, DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00068. **Core Rank B**.

C4 Peini Liu, Jordi Guitart and Amir Taherkordi, "Performance Characterization of Multi-container Deployment Schemes for Online Machine Learning Inference on Kubernetes Clusters", *2023 IEEE International Conference on Cloud Computing (CLOUD)*, July 2023, Chicago, USA. Accepted. **Core Rank B**.

Publications not included in this thesis

Published conference papers

C5 Peini Liu, Gusseppe Bravo-Rocca and Jordi Guitart, "Energy-aware Dynamic Pricing Model for Cloud Environments", *The 16th International Conference on the Economics of Grids, Clouds, Systems, and Service*, September 2019, Leeds, United Kingdom, proceedings, vol 11819, pp. 1-10, DOI: 10.1007/978-3-030-36027-6_7.

C6 Peini Liu and Jordi Guitart, "An architecture for automatic ML/AI workflow management and supervision", *Barcelona Supercomputing Center 8th Doctoral Symposium*, May 2021, Barcelona, Spain, pp. 44-45.

Publications collaborated

Published journal papers

J3 Gusseppe Bravo-Rocca, Peini Liu, Jordi Guitart, Ajay Dholakia, David Ellison, Jeffrey Falkanger and Miroslav Hodak, "Scanflow: A multi-graph framework for Machine Learning workflow management, supervision, and debugging", *Expert Systems with Applications*, vol. 202, pp. 117232, issn. 0957-4174, September 2022. DOI: 10.1016/j.eswa.2022.117232. **JCR Q1**.

Published conference papers

C7 Gusseppe Bravo-Rocca, Peini Liu, Jordi Guitart, Ajay Dholakia, David Ellison and Miroslav Hodak, "Human-in-the-loop online multi-agent approach to increase trustworthiness in ML models through trust scores and data augmentation", *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, June 2022, Virtual Event, USA, pp. 32-37, DOI: 10.1109/COMPSAC54236.2022.00014. **Core Rank B**.

Other Contributions

O1 May 29, 2021 - Software Release on the Github repository: "Scanflow-Kubernetes: An MLOps Platform". Available at: `https://github.com/bsc-scanflow/scanflow`.

- M1: (26/06/2021) Release master v0.1.0 Scanflow-Kubernetes basic.
- M2: (13/12/2021) Release master v0.1.1 Scanflow-Kubernetes with resource, affinity and HPA definition.
- M3: (27/05/2022) Release mpi branch Scanflow(MPI)-Kubernetes enhanced MPI applications support.

O2 May, 2021 - Contribute to CNCF project Couler with functions to support specifying volume mount.

- M1: (22/05/2021) A pull request “Fix issue with volume mount” is merged. Available at: https://github.com/couler-proj/couler/pull/194.

O3 Nov, 2021 - Contribute to CNCF project Volcano with functional enhancements.

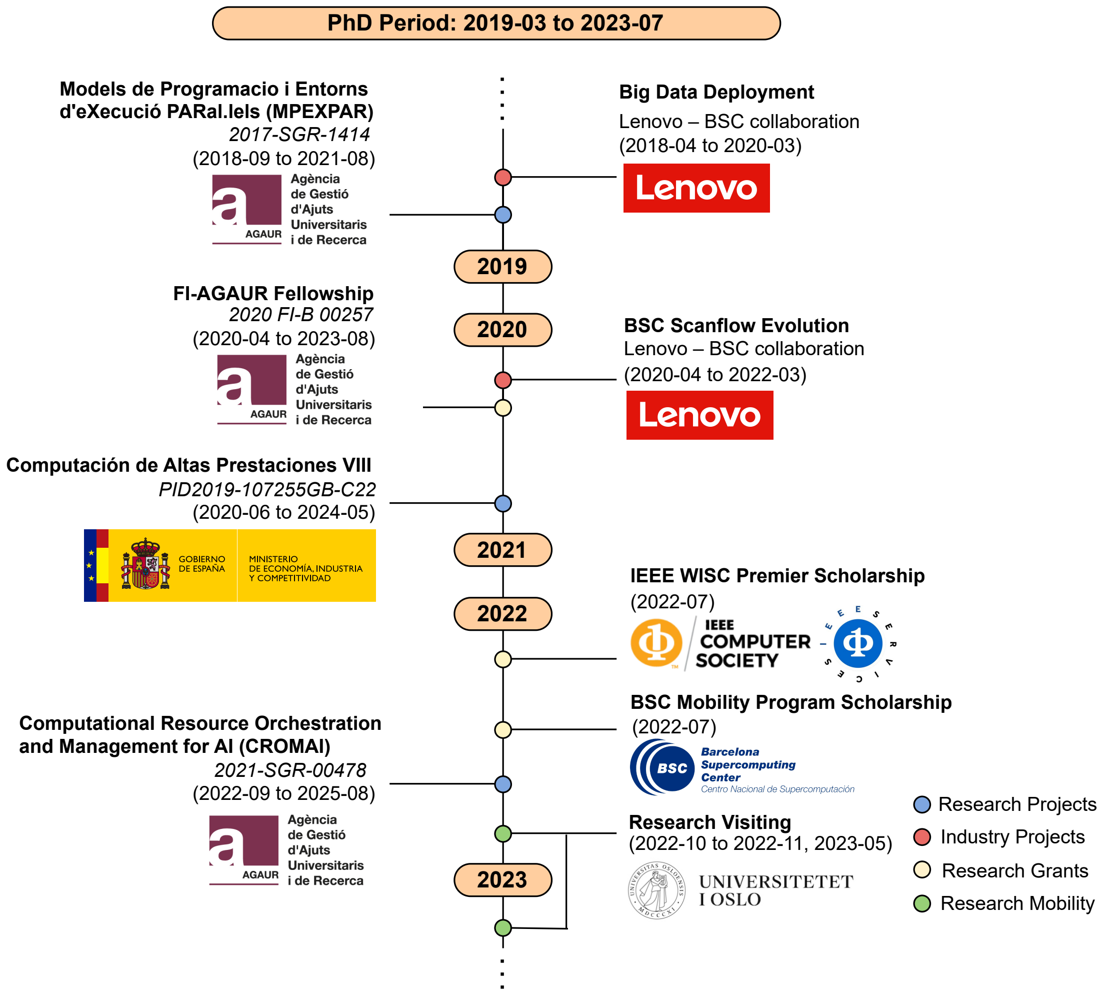
- M1: (18/07/2022) A pull request “Add GPU Numbers Support” for volcano device plugin is merged. Available at: https://github.com/volcano-sh/devices/pull/15.
- M2: (15/08/2022) A pull request “Add GPU Numbers Predicates” for volcano scheduler is merged. Available at: https://github.com/volcano-sh/volcano/pull/1692.
- M3: (17/10/2022) A pull request “Enhance doc” for volcano device plugin is merged. Available at: https://github.com/volcano-sh/devices/pull/27.
- M4: (17/10/2022) A pull request “support config gpu memory factor” for volcano device plugin is merged. Available at: https://github.com/volcano-sh/devices/pull/28.

O4 Jan 11, 2022 - obtain CKA(Certified Kubernetes Administrator) issued by The Linux Foundation.

Appendix B

Projects, Grants and Activities

This section presents a summary of the projects, grants, and relevant activities that the author has been involved during the PhD time period. A timeline with the projects, grants, and main activities carried out by the author during her PhD time period is shown in Figure B.1. Other detailed research activities she has attended, including courses, seminars, and conferences, are listed in the following sections.



PhD Period: 2019-03 to 2023-07
Models de Programacio i Entorns d'eXecució PARal.lels (MPEXPAR)
2017-SGR-1414
(2018-09 to 2021-08)
Agència de Gestió d'Ajuts Universitaris i de Recerca
AGAUR
Big Data Deployment
Lenovo – BSC collaboration
(2018-04 to 2020-03)
Lenovo
2019
FI-AGAUR Fellowship
2020 FI-B 00257
(2020-04 to 2023-08)
2020
BSC Scanflow Evolution
Lenovo – BSC collaboration
(2020-04 to 2022-03)
Computación de Altas Prestaciones VIII
PID2019-107255GB-C22
(2020-06 to 2024-05)
GOBIERNO DE ESPAÑA
MINISTERIO DE ECONOMÍA, INDUSTRIA Y COMPETITIVIDAD
2021
IEEE WISC Premier Scholarship
(2022-07)
IEEE COMPUTER SOCIETY
2022
BSC Mobility Program Scholarship
(2022-07)
Barcelona Supercomputing Center
Centro Nacional de Supercomputación
Computational Resource Orchestration and Management for AI (CROMAI)
2021-SGR-00478
(2022-09 to 2025-08)
Research Visiting
(2022-10 to 2022-11, 2023-05)
UNIVERSITETET I OSLO
2023
● Research Projects
● Industry Projects
● Research Grants
● Research Mobility

Figure B.1: Timeline of the projects, grants and main activities carried out during the PhD time period.

Grants

- Obtained grant from Generalitat de Catalunya under number 2020 FI-B 00257.
- Obtained 2022 IEEE WISC Premier Scholarship announced at https://conferences.computer.org/services/2022/symposia/wisc_symposium.html.
- Obtained BSC mobility program grant announced at https://www.bsc.es/join-us/why-to-work-at-bsc/mobility-programmes/bsc-mobility-program.

Mobility

- Research Visiting with Amir Taherkordi from the Network and Distributed Systems Group at the Department of Informatics at the University of Oslo (UiO), Norway, Oct - Nov 2022, May 2023.

Attendance at a course, seminar or conference

- Attended and presented a paper at 16th International Conference on the Economics of Grids, Clouds, Systems and Services on Sep 2019.
- Attended “Parallel Programming Workshop of PATC@BSC” Training Course at Barcelona Supercomputing Center (BSC) on $14^{th} - 18^{th}$ Oct 2019.
- Attended “Big Data Analytics of PATC@BSC” Training Course at Barcelona Supercomputing Center on $3^{rd} - 7^{th}$ Feb 2020.
- Attended “Responsible Conduct in Research and Innovation” Course at Universitat Politècnica de Catalunya (UPC) on fall semester (1^{st} Aug-16^{th} Oct 2020).
- Attended “The road to competitive funding” Training Course at Barcelona Supercomputing Center on $21^{st} - 23^{rd}$ Oct 2020.
- Attended “MLOps (Machine Learning Operations) Fundamentals” Course funded by Google Cloud on Jan 2021.
- Attended “KubeCon + CloudNativeCon Europe 2021 Virtual/Kubernetes AI Day hosted by CNCF + LF AI” Conference organized by CNCF (Cloud Native Computing Foundation) on $4^{th} - 7^{th}$ May 2021.
- Attended, presented a paper, and collaborated as an volunteer at 8th BSC Doctoral Symposium organized by Barcelona Supercomputing Center on $11^{th} - 13^{th}$ May 2021.
- Attended ACM Summer School on HPC Computer Architectures for AI and Dedicated Applications on 30^{st} Aug - 3^{th} Sep 2021.
- Attended and presented a demo paper at The 22nd ACM/IFIP International Middleware Conference on Dec 2021.
- Attended and presented a paper at The 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing on May 2022.

- Attended international conference IEEE World Congress on SERVICES on Jul 2022 and gained IEEE WISC scholarship at 2022 IEEE international symposium on women in service computing (WISC 2022) on Jul 2022.
- Attended and presented a paper at The 2022 High Performance Computing and Communications (HPCC2022) and was a session chair on Dec 2022.
- Presented one hour talk at BSC research seminar with the topic "Convergence of HPC, Big Data and Machine Learning Applications on Containerized Infrastructures" on 09 Mar 2023. https://www.bsc.es/research-and-development/research-seminars/hybrid-sorswics-convergence-hpc-big-data-and-machine-learning-applications-and-containerized

Bibliography

[1] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, et al. Early evaluation of IBM BlueGene/P. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*, pages 1–12. IEEE, 2008. 15

[2] A. Ali, R. Pinciroli, F. Yan, and E. Smirni. Batch: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *Proc. of the Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, SC'20. IEEE Press, 2020. 106

[3] Apache Hadoop. Apache hadoop. 5

[4] Apache Hadoop. Hdfs architecture. 17

[5] Apache Hadoop. Mapreduce tutorial. 4, 16

[6] Apache Spark. Apache spark. 5

[7] Apache Spark. Running spark on kubernetes. 31

[8] C. Arango, R. Dernat, and J. Sanabria. Performance evaluation of container-based virtualization for high performance computing environments. *arXiv preprint arXiv:1709.10140*, 2017. 24, 26, 36, 37

[9] M. Asch, T. Moore, R. Badia, M. Beck, P. Beckman, T. Bidot, F. Bodin, F. Cappello, A. Choudhary, B. de Supinski, E. Deelman, J. Dongarra, A. Dubey, G. Fox, H. Fu, S. Girona, W. Gropp, M. Heroux, Y. Ishikawa, K. Keahey, D. Keyes, W. Kramer, J. F. Lavignon, Y. Lu, S. Matsuoka, B. Mohr, D. Reed, S. Requena, J. Saltz, T. Schulthess, R. Stevens, M. Swany, A. Szalay, W. Tang, G. Varoquaux, J. P. Vilotte, R. Wisniewski, Z. Xu, and I. Zacharov. *Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry*, volume 32. 2018. 1, 2, 11, 129

[10] G. Aupy, A. Gainaru, V. Honoré, P. Raghavan, Y. Robert, and H. Sun. Reservation strategies for stochastic jobs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 166–175, 2019. 34

[11] A. Azab. Enabling Docker Containers for High-Performance and Many-Task Computing. In *Proceedings of the 2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 279–285, April 2017. 37

[12] Azure. Azure Kubernetes Service (AKS), 2022. 129

[13] J. Bacik. Cpu scheduler imbalance with cgroups. 49

[14] A. Banerjee, R. Mehta, and Z. Shen. NUMA Aware I/O in Virtualized Systems. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 10–17, 2015. 25

[15] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. In *Proceedings of CANOPIE-HPC 2019: 1st International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC*, pages 11–20, 2019. 27, 129

[16] B. Bermejo and C. Juiz. On the classification and quantification of server consolidation overheads. *Journal of Supercomputing*, pages 1–21, mar 2020. 25

[17] J. Bhimani, Z. Yang, M. Leeser, and N. Mi. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017. 4, 23, 24

[18] Big data interagency working group and the high end computing interagency working group, Networking & information technology research & development subcommittee, Committee on science & technology enterprise of the National Science and Technology Council. THE CONVERGENCE OF HIGH PERFORMANCE COMPUTING, BIG DATA, AND MACHINE LEARNING. September 2019. 1, 2, 34, 129

[19] B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Fréchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, and J. Vanschoren. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016. 19, 32, 105

[20] G. Bravo-Rocca, P. Liu, J. Guitart, A. Dholakia, D. Ellison, J. Falkanger, and M. Hodak. Scanflow: A multi-graph framework for machine learning workflow management, supervision, and debugging, 2021. 33, 112, 121

[21] F. M. Brazier, J. O. Kephart, H. V. D. Parunak, and M. N. Huhns. Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda. *IEEE Internet Computing*, 13(3):82–87, 2009. 33, 106

[22] P. Carpenter, M. Casas, O. Unsal, P. Radojkovic, X. Martorell, A. Miranda, J. Guitart, J. Corbalan, A. Peña, L. A. Bautista Gomez, and et al. *ETP4HPC's SRA 5 strategic research agenda for High-Performance Computing in Europe 2022: European HPC research priorities 2023-2027*. Sep 2022. 1

[23] Y. Cheng, W. Chen, X. Chen, B. Xu, and S. Zhang. A user-level numa-aware scheduler for optimizing virtual machine performance. In *Revised Selected Papers of the 10th International Symposium on Advanced Parallel Processing Technologies - Volume 8299*, APPT 2013, pages 32–46, Berlin, Heidelberg, 2013. Springer-Verlag. 25

[24] M. T. Chung, A. Le, N. Quang-Hung, D. Nguyen, and N. Thoai. Provision of Docker and InfiniBand in High Performance Computing. In *Proceedings of the 2016*

International Conference on Advanced Computing and Applications, ACOMP'16, pages 127–134. IEEE, 2016. 27, 37

[25] M. T. Chung, N. Quang-Hung, M. Nguyen, and N. Thoai. Using Docker in High Performance Computing applications. In *Proceedings of the 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57, July 2016. 26, 27, 36, 37

[26] P. Covington, J. Adams, and E. Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 191–198. Association for Computing Machinery, 2016. 2, 17, 89

[27] C. Cox, D. Sun, E. Tarn, A. Singh, R. Kelkar, and D. Goodwin. Serverless inferencing on Kubernetes, 2020. 106

[28] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 613–627. USENIX Association, mar 2017. 19, 32

[29] J. D. Fast virtualized Hadoop and Spark on all-flash disks. Technical report, 2017. 4, 22

[30] T. De Wolf and T. Holvoet. Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, pages 470–479, 2003. 33, 106

[31] Docker. Docker swarm strategies. 29

[32] Docker. Swarm filters. 29

[33] Docker. Deploy to Swarm, 2022. 129

[34] R. Farber. AI-HPC is Happening Now. Technical report, 2017. 2

[35] D. G. Feitelson. Metrics for parallel job scheduling and their convergence. In *Job Scheduling Strategies for Parallel Processing (JSSPP'01)*, pages 188–205. Springer, 2001. 140

[36] D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing (JSSPP'98)*, pages 1–24. Springer, 1998. 140

[37] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software*, pages 171–172, 2015. 2, 22, 35, 36

[38] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and Robust Automated Machine Learning. In *Proc. of the 28th Intl. Conference on Neural Information Processing Systems - Vol. 2*, NIPS'15, pages 2755–2763. MIT Press, 2015. 19, 32, 105

[39] L. Foster. The mpi programming model. 5, 13

[40] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve. Big data, simulations and hpc convergence. In *Big Data Benchmarking*, pages 3–17. Springer, 2015. 1

[41] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao. Progress-based container scheduling for short-lived applications in a kubernetes cluster. In *IEEE Intl. Conference on Big Data*, pages 278–287, 2019. 34

[42] J. Gama, I. Žliobaitundefined, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.*, 46(4), Mar. 2014. 33, 106

[43] Gartner. Aiops (artificial intelligence for it operations). 5

[44] Gartner. Aiops platform. 5

[45] Gartner. Big data. 4, 15

[46] Google. Cgroups-cpus. 49, 100

[47] Google. Google Kubernetes Engine (GKE), 2022. 129

[48] Google Cloud. Machine learning workflow, 2021. 18, 105

[49] Google Cloud. MLOps: Continuous delivery and automation pipelines in machine learning, 2021. xv, 18, 106

[50] P. Grun. Introduction to Infiniband for end users. Technical report, InfiniBand Trade Association, 2010. 14

[51] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg. Swayam: Distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware'17, pages 109–120. Association for Computing Machinery, 2017. 28, 90

[52] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B.-S. Lee, V. March, D. Milojicic, and C. H. Suen. Evaluating and improving the performance and scheduling of hpc applications in cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321, 2016. 34, 129

[53] G. Halácsy and Z. A. Mann. Optimal energy-efficient placement of virtual machines with divisible sizes. *Information Processing Letters*, 138:51–56, 2018. 36

[54] N. Hasabnis. Auto-tuning tensorflow threading model for cpu backend. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 14–25. IEEE Computer Society, nov 2018. 19, 90

[55] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015. 2, 17, 20, 89, 94

[56] M. Heller. The best machine learning and deep learning libraries. 17

[57] M. Herman. Deploying spark on kubernetes. 17

[58] A. J. Hey, S. Tansley, K. M. Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009. 4, 11, 15

[59] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for Fine-Grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011. 129

[60] T. Hoefler, W. Gropp, R. Thakur, and J. L. Träff. Toward performance models of mpi implementations for understanding application scaling issues. In *Recent Advances in the Message Passing Interface (EuroMPI 2010)*, pages 21–30. Springer, 2010. 138

[61] HPC Advisory Council. Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing, 2009. White paper. 84

[62] HPC Advisory Council. HPCC Performance Benchmark and Profiling, 2015. 15, 37, 68

[63] HPC wire. Sylabs releases singularity 3.0 container platform; Cites AI Support, October 2018. 24

[64] E. A. Huerta, A. Khan, E. Davis, C. Bushell, W. D. Gropp, D. S. Katz, V. Kindratenko, S. Koric, W. T. C. Kramer, B. McGinty, K. McHenry, and A. Saxton. Convergence of artificial intelligence and high performance computing on NSF-supported cyberinfrastructure. *Journal of Big Data*, 7(1), oct 2020. 2

[65] IBM. Documentation update: Gss 2.0 information (applied to gpfs version 4 release 1 information units). 11

[66] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu. Characterizing the performance of parallel applications on multi-socket virtual machines. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1–12. IEEE, 2011. 25

[67] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu. The Case for Partitioning Virtual Machines on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2683–2696, 2014. 25, 36, 37, 55, 56

[68] A. Imbrea. *An empirical comparison of automated machine learning techniques for data streams*. PhD thesis, January 2020. 33

[69] Intel. Lustre * Performance is Superior to HDFS * with the Latest Intel ® Xeon ® Processor Family. pages 3–6. 4

[70] Intel. Bare-metal performance for Big Data workloads on Docker Containers. Technical report, 2017. 4, 23, 24

[71] Intel. Math kernel library, 2018. 19, 32

[72] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, 2011. 35, 36

[73] B. J. Virtualized Hadoop Performance with VMware vSphere 6 on High-Performance Servers. Technical report, 2015. 4, 22

[74] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, G. Morgan, and R. Ranjan. A study on the evaluation of HPC microservices in containerized environment. *Concurrency and Computation*, (March):1–18, 2019. 26, 36

[75] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan. A Holistic Evaluation of Docker Containers for Interfering Microservices. In *Proceedings of the 2018 IEEE International Conference on Services Computing (SCC)*, pages 33–40, 2018. 26, 36

[76] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. Fox. Low latency stream processing: Apache heron with infiniband & intel omni-path. In *Proceedings of the 10th International Conference on Utility and Cloud Computing*, pages 101–110. ACM, 2017. 4

[77] G. Katz, E. C. R. Shin, and D. Song. ExploreKit: Automatic Feature Generation and Selection. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 979–984, 2016. 19, 32, 106

[78] D. J. Kedziora, K. Musial, and B. Gabrys. AutonoML: Towards an Integrated Framework for Autonomous Machine Learning, 2020. 32, 106

[79] G. Kim, K. Behr, and G. Spafford. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win.* IT Revolution Press, 1st edition, 2013. 5

[80] J. Klaise, A. V. Looveren, C. Cox, G. Vacanti, and A. Coca. Monitoring and explainability of models in production, 2020. 33

[81] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017. 2, 17, 89

[82] Kserve. Highly scalable and standards based model inference platform on kubernetes for trusted AI, 2021. 19, 32

[83] Kubeflow. Kubeflow MPI Training (MPIJob), 2021. 31, 130, 131, 144

[84] Kubernetes. Configure multiple schedulers. 30

[85] Kubernetes. Kubernetes scheduler. 30

[86] Kubernetes. Scheduler performance tuning. 30

[87] Kubernetes. Scheduling framework. 29, 30

[88] Kubernetes. Why you need Kubernetes and what it can do, 2021. 125

[89] Kubernetes. Production-Grade Container Orchestration, 2022. 129

[90] A. Kuity and S. K. Peddoju. Performance Evaluation of Container-Based High Performance Computing Ecosystem Using OpenPOWER. In J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, editors, *High Performance Computing, ISC High Performance 2017, Lecture Notes in Computer Science, vol. 10524*, pages 290–308, Cham, 2017. Springer International Publishing. 2, 35

[91] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017. 24, 37

[92] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626. USENIX Association, oct 2018. 4, 18, 28, 89, 90

[93] L. Li, J. Chen, and W. Yan. A particle swarm optimization-based container scheduling algorithm of docker platform. In *Proceedings of the 4th International Conference on Communication and Information Processing*, ICCIP â18, page 12â17, New York, NY, USA, 2018. Association for Computing Machinery. 30

[94] Q. Li and Y. Fang. Multi-algorithm collaboration scheduling strategy for docker container. *2017 International Conference on Computer Systems, Electronics and Control (ICCSEC)*, pages 1367–1371, 2017. 30

[95] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. 2

[96] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang. A new container scheduling algorithm based on multi-objective optimization. *Soft Computing*, 22(23):7741–7752, Dec 2018. 30

[97] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak. Scanflow: An end-to-end agent-based autonomic ml workflow manager for clusters. In *Proceedings of the 22nd International Middleware Conference: Demos and Posters*, Middleware '21, page 1â2, New York, NY, USA, 2021. Association for Computing Machinery. 4, 6, 18, 89, 130

[98] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak. Scanflow-k8s: Agent-based framework for autonomic management and supervision of ML workflows in kubernetes clusters. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 376–385, 2022. 4, 7, 8, 18, 89, 130

[99] P. Liu and J. Guitart. Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study. *Journal of Supercomputing*, 77:6273–6312, 2020. 6, 34, 90, 130, 131, 132, 140

[100] P. Liu and J. Guitart. Fine-grained scheduling for containerized hpc workloads in kubernetes clusters. In *2022 IEEE 24th International Conference on High Performance Computing and Communications (HPCC)*, pages 275–284, Dec 2022. 7, 8

[101] P. Liu and J. Guitart. Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study. *Cluster Computing*, 25:847–868, 2022. 6, 34, 90, 130, 131, 132

[102] P. Liu and J. Guitart. Performance characterization of multi-container deployment schemes for online learning inference, 2022. 6

[103] P. Liu, X. Mao, S. Zhang, and F. Hou. Towards reference architecture for a multi-layer controlled self-adaptive microservice system. In *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 236–241, 2018. 33, 106

[104] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, oct 2014. 28, 90

[105] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSysâ16. Association for Computing Machinery, 2016. 49, 63

[106] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang. Learning under Concept Drift: A Review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2019. 33, 106

[107] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda. Accelerating spark with RDMA for big data processing: Early experiences. *Proceedings - 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects, HOTI 2014*, pages 9–16, 2014. xv, 14

[108] X. Lu, D. Shankar, S. Gugnani, and D. K. D. Panda. High-performance design of apache spark with rdma and its benefits on various workloads. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 253–262. IEEE, 2016. 4

[109] X. Lu, D. Shankar, H. Shi, and D. K. Dk Panda. Spark-uDAPL: Cost-Saving Big Data Analytics on Microsoft Azure Cloud with RDMA Networks. *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, pages 321–326, 2019. 14

[110] X. Lu, M. Wasi-ur Rahman, N. Islam, D. Shankar, and D. K. D. Panda. *Accelerating Big Data Processing on Modern HPC Clusters*, pages 81–107. Springer International Publishing, Cham, 2016. xv, 14

[111] T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, Sept. 2015. Association for Computational Linguistics. 2, 17, 89

[112] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, pages 213–es. ACM, 2006. 14, 68

[113] Luszczek, P. and Koester, D. HPC Challenge v1.x Benchmark Suite. SC'05 Tutorial, Seattle, Washington, 2005. 40

[114] A. M. Maliszewski, D. Griebler, C. Schepke, A. Ditter, D. Fey, and L. G. Fernandes. The NAS Benchmark Kernels for Single and Multi-Tenant Cloud Instances with LXC/KVM. In *Proceedings of the 2018 International Conference on High Performance Computing Simulation (HPCS)*, pages 359–366, July 2018. 26, 36

[115] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947. 41

[116] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. 2

[117] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana. Client-side scheduling based on application characterization on kubernetes. In C. Pham, J. Altmann, and J. Á. Bañares, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 162–176, Cham, 2017. Springer International Publishing. 30

[118] V. Medel, R. Tolosana, J. A. B. nares, U. Arronategui, and O. F. Rana. Characterising resource management performance in Kubernetes. *Computers & Electrical Engineering*, 68:286–297, 2018. 28, 34

[119] Mellanox Technologies. Introducing 200G HDR InfiniBand Solutions White Paper. pages 1–4. 13

[120] Mellanox Technologies. Top 500 High Performance Computing Platform Interconnect. 13

[121] Mellanox Technologies. Introduction to InfiniBand. *Technical Report*, pages 1–20, 2003. 13

[122] Mellanox Technologies. CASE STUDY HIGHLIGHTS ABOUT SUMMIT. Technical report, 2018. 13

[123] Mellanox Technologies. Interconnect Your Future Enabling the Best Datacenter Return on Investment. Technical report, 2019. 13

[124] T. Menouer, O. Manad, C. Cérin, and P. Darmon. Power efficiency containers scheduling approach based on machine learning technique for cloud computing environment. In C. Esposito, J. Hong, and K.-K. R. Choo, editors, *Pervasive Systems, Algorithms and Networks*, pages 193–206, Cham, 2019. Springer International Publishing. 30

[125] M. Mercier, D. Glesser, Y. Georgiou, and O. Richard. Big data and HPC collocation: Using HPC idle resources for Big Data analytics. In *2017 IEEE Intl. Conference on Big Data (Big Data)*, pages 347–352, 2017. 34

[126] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11, Apr. 2006. 22

[127] C. Misale, M. Drocco, D. J. Milroy, C. E. A. Gutierrez, S. Herbein, D. H. Ahn, and Y. Park. It's a Scheduling Affair: GROMACS in the Cloud with the KubeFlux Scheduler. In *2021 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 10–16, 2021. 34

[128] Morgan Funtowicz. Scaling up BERT-like model Inference on modern CPU - Part 1, April 2021. 28

[129] Morgan Funtowicz. Scaling up BERT-like model Inference on modern CPU - Part 2, November 2021. 28

[130] NASA. NASA Global Weather Forecasting Jumps Forward — NASA Center for Climate Simulation. 4, 11

[131] National Academies of Sciences, Engineering, and Medicine. *Future Directions for NSF Advanced Computing Infrastructure to Support U.S. Science and Engineering in 2017-2020*. The National Academies Press, Washington, DC, 2016. 1

[132] National Academies of Sciences, Engineering, and Medicine. *Opportunities from the Integration of Simulation Science and Data Science: Proceedings of a Workshop*. The National Academies Press, Washington, DC, 2018. 1

[133] Nvidia. HPC and AI, 2020. 2

[134] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. TensorFlow-Serving: Flexible, High-Performance ML Serving, 2017. 5, 106

[135] OpenMPI Team. Can i force aggressive or degraded performance modes? 13, 53

[136] OpenMPI Team. Can I oversubscribe nodes (run more processes than processors)? 13, 53

[137] D. K. Panda. Challenges and Opportunities in Designing High-Performance and Scalable Middleware for HPC and AI: Past, Present, and Future, May 2022. Remarks by Dhabaleswar K. Panda at 36th IEEE International Parallel Distributed Processing Symposium. 2

[138] D. K. Panda. Designing Next-Generation Intelligent CyberInfrastructure: An Overview of the NSF-AI ICICLE Institute, June 2022. Remarks by Dhabaleswar K. Panda at NIRTD MAGIC Seminar Series. 1

[139] D. K. Panda and X. Lu. Hpc meets cloud: Building efficient clouds for hpc, big data, and deep learning middleware and applications. In *Proceedings of the 10th International Conference on Utility and Cloud Computing*, pages 189–190. ACM, 2017. 1

[140] C. Park and S. Paul. Load-testing TensorFlow Servingâs REST Interface, July 2022. 28

[141] S. Perarnau, B. C. V. Essen, R. Gioiosa, K. Iskra, M. B. Gokhale, K. Yoshii, and P. Beckman. Argo. *Operating Systems for Supercomputers and High Performance Computing*, 2019. 56

[142] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of the 18th World Occam and Transputer User Group Technical Meeting*, pages 9–13. IOS Press, 1995. 38

[143] Pytorch. FROM RESEARCH TO PRODUCTION: An open source machine learning framework that accelerates the path from research prototyping to production deployment. 2, 5, 19, 32

[144] PyTorch. Torchserver, 2020. 19, 32, 89

[145] M. Rahman, X. Lu, N. S. Islam, R. Rajachandrasekar, D. K. Panda, et al. Mapreduce over lustre: Can rdma-based approach benefit? In *European Conference on Parallel Processing*, pages 644–655. Springer, 2014. 4

[146] M. W. U. Rahman, N. S. Islam, X. Lu, and D. K. Panda. A Comprehensive Study of MapReduce over Lustre for Intermediate Data Placement and Shuffle Strategies on HPC Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):633–646, 2017. 4

[147] A. Raj, K. Kaur, U. Dutta, V. V. Sandeep, and S. Rao. Enhancement of hadoop clusters with virtualization using the capacity scheduler. In *2012 Third International Conference on Services in Emerging Markets*, pages 50–57, Dec 2012. 4, 22

[148] J. Rao, K. Wang, X. Zhou, and C. Xu. Optimizing virtual machine scheduling in NUMA multicore systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 306–317, 2013. 25

[149] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020. xv, 20, 21, 91, 95

[150] D. A. Reed and J. Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, June 2015. 11

[151] G. Rimassa. Towards a vibrant European Cloud Computing ecosystem, Oct 2022. 2

[152] G. Rimassa and F. M.Facca. Digital Autonomy in the Computing Continuum, Nov 2021. 2

[153] E. Roloff, M. Diener, A. Carissimi, and P. O. A. Navaux. High Performance Computing in the cloud: Deployment, performance and cost efficiency. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science*, pages 371–378, 2012. 35

[154] O. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vázquez. Containers in HPC: A Scalability and Portability Study in Production Biological Simulations. In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 567–577, 2019. 26, 27

[155] Run.AI. Machine Learning Workflow: Automating Machine Learning Workflows, 2021. 19, 32, 105

[156] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. 2

[157] P. Saha, A. Beltre, and M. Govindaraju. Scylla: A mesos framework for container based MPI jobs. *CoRR*, abs/1905.08386, may 2019. 26, 27, 34, 36

[158] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju. Evaluation of Docker Containers for Scientific Workloads in the Cloud. In *Proceedings of the Practice and Experience on Advanced Research Computing*, PEARC'18. Association for Computing Machinery, 2018. 26, 27, 34, 36, 37

[159] V. Sande Veiga, M. Simon, A. Azab, C. Fernandez, G. Muscianisi, G. Fiameni, and S. Marocchi. Evaluation and Benchmarking of Singularity MPI containers on EU Research e-Infrastructure. In *Proceedings of the 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 1–10, 2019. 24

[160] C. Sauvanaud, A. Dholakia, J. Guitart, C. Kim, and P. Mayes. Big data deployment in containerized infrastructures through the interconnection of network namespaces. *Software: Practice and Experience*, 50(7):1087–1113, 2020. 6, 17, 25, 27, 68

[161] Seldon. Machine learning deployment for enterprise, 2021. 5, 19, 32

[162] A. Shahraki, M. Abbasi, A. Taherkordi, and A. D. Jurcut. A comparative study on online machine learning techniques for network traffic streams analysis. *Computer Networks*, 207:108836, 2022. 90

[163] T. Shanley. *InfiniBand Network Architecture*. Addison Wesley, 2002. 13

[164] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 12 1965. 41

[165] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. Containers and Virtual Machines at Scale. *Proceedings of the 17th International Middleware Conference on - Middleware '16*, pages 1–13, 2016. 21, 22, 25, 36

[166] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 322–337. Association for Computing Machinery, 2019. 28

[167] Slurm. Slurm Workload Manager, 2022. 34

[168] Slurm. Slurm Workload Manager - Containers Guide, 2022. 34

[169] Spark. Apache spark on kubernetes. 17

[170] Spark. Running spark on kubernetes. 17

[171] Spark. Running spark on mesos. 17

[172] Spark. Running spark on yarn. 17

[173] T. Sterling, M. Anderson, and M. Brodowicz. The Essential Resource Management. In *High Performance Computing, Chapter 5*, pages 141–190. Morgan Kaufmann, Boston, 2018. 38

[174] H. Subramoni, P. Lai, M. Luo, and D. K. Panda. RDMA over Ethernet: A preliminary study. In *Proc. of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09)*, pages 1–9, 2009. 13, 14

[175] Tensorflow. Create production-grade machine learning models with TensorFlow. 2, 5, 19, 32

[176] TensorFlow. Tensorflow - serving models, 2021. 5, 19, 32, 89

[177] G. Tesauro, D. Chess, W. Walsh, R. Das, A. Segal, I. Whalley, J. Kephart, and S. White. A multi-agent systems approach to autonomic computing. In *Proc. of the 3rd Intl. Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 464–471, 2004. 33, 106

[178] S. K. Tesfatsion, C. Klein, and J. Tordsson. Virtualization Techniques Compared: Performance, Resource, and Power Usage Overheads in Clouds. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE â18, pages 145–156. Association for Computing Machinery, 2018. 25

[179] R. Todling, A. E. Akkraoui, and R. D. Koster. Technical Report Series on Global Modeling and Data Assimilation, Volume 50 The GMAO Hybrid Ensemble-Variational Atmospheric Data Assimilation System: Version 2.0. Technical report, 2018. 11

[180] Top500. Highlights — TOP500 Supercomputer Sites. 13

[181] A. Torrez, T. Randles, and R. Priedhorsky. HPC Container Runtimes have Minimal or No Performance Impact. In *Proceedings of the 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 37–42, 2019. 36

[182] B. M. Tudor and Y. M. Teo. A Practical Approach for Performance Analysis of Shared-Memory Programs. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium*, pages 652–663, 2011. 53, 75

[183] A. Uta, A. L. Varbanescu, A. Musaafir, C. Lemaire, and A. Iosup. Exploring HPC and Big Data Convergence: A Graph Processing Study on Intel Knights Landing. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 66–77, 2018. 2, 4

[184] Vmware. Virtualized high performance computing (hpc) reference architecture (part 1 of 2). 22

[185] Vmware. VIRTUALIZING HIGH-PERFORMANCE COMPUTING (HPC) ENVIRONMENTS Reference Architecture. (SEPTEMBER), 2018. 22, 25, 38

[186] Volcano. MPI on Volcano, 2021. 30, 32, 130, 144

[187] R. W. Collier, E. O'Neill, D. Lillis, and G. O'Hare. MAMS: Multi-Agent MicroServices. In *Proceedings of the 2019 World Wide Web Conference*, WWW'19, pages 655–662. ACM, 2019. 112

[188] R. Walkup, S. R. Seelam, and S. Wen. Best Practices for HPC Workloads on Public Cloud Platforms: A Guide for Computational Scientists to Use Public Cloud for HPC Workloads. In *ACM/SPEC Intl. Conference on Performance Engineering (ICPE)*, pages 29–35, 2022. 34, 129

[189] L. Wang, L. Yang, Y. Yu, W. Wang, B. Li, X. Sun, J. He, and L. Zhang. Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'21, pages 639–653. ACM, 2021. 33

[190] S. Wang, Z. Ding, and C. Jiang. Elastic scheduling for microservice applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(01):98–115, jan 2021. 28

[191] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad. Rafiki: Machine learning as an analytics service system. *Proc. VLDB Endow.*, 12(2):128–140, oct 2018. 19, 32

[192] Y. Wang, R. T. Evans, and L. Huang. Performant Container Support for HPC Applications. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARCâ19, pages 1–6. Association for Computing Machinery, 2019. 26

[193] B. L. Welch. The Generalization of Student's Problem When Several Different Population Variances Are Involved. *Biometrika*, 34(1-2):28–35, 01 1947. 41

[194] T. White. *Hadoop: The definitive guide.* xv, 4, 15, 16

[195] Wikipedia. Tensor Processing Unit. 2

[196] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, Feb 2013. 26, 36, 37

[197] F. Xing, H. You, and C. Lu. HPC benchmark assessment with statistical analysis. *Procedia Computer Science*, 29:210–219, 2014. 14, 68

[198] S. Yang, X. Wang, L. An, and G. Zhang. Yun: A High-Performance Container Management Service Based on OpenStack. In *Proceedings of the 2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, pages 202–209, 2019. 25

[199] Q. Yao, M. Wang, Y. Chen, W. Dai, Y.-F. Li, W.-W. Tu, Q. Yang, and Y. Yu. Taking Human out of Learning Applications: A Survey on Automated Machine Learning, 2019. 19, 32, 105

[200] K. Ye and Y. Ji. Performance tuning and modeling for big data applications in docker containers. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6, Aug 2017. 4, 23, 24

[201] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox. Analysis of virtualization technologies for high performance computing environments. *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, pages 9–16, 2011. 22

[202] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell. A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds. In *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 74–81, Dec 2017. 2, 26, 35, 36

[203] C. Zhang, M. Yu, W. Wang, and F. Yan. Enabling cost-effective, SLO-aware machine learning inference serving on public cloud. *IEEE Transactions on Cloud Computing*, 10(3):1765–1779, 2022. 4, 18, 28, 89, 90

[204] J. Zhang, X. Lu, and D. K. Panda. High performance MPI library for container-based HPC cloud on InfiniBand clusters. In *45th International Conference on Parallel Processing (ICPP)*, pages 268–277. IEEE, 2016. 21, 27, 36, 37, 65

[205] J. Zhang, X. Lu, and D. K. Panda. Performance characterization of hypervisor-and container-based virtualization for HPC on SR-IOV enabled infiniband clusters. In *Proc. of 30th Intl. Parallel and Distributed Processing Symposium (IPDPS'16)*, pages 1777–1784. IEEE, 2016. 27

[206] J. Zhang, X. Lu, and D. K. Panda. Designing Locality and NUMA Aware MPI Runtime for Nested Virtualization Based HPC Cloud with SR-IOV Enabled InfiniBand. *SIGPLAN Not.*, 52(7):187–200, Apr. 2017. 27, 37

[207] J. Zhang, X. Lu, and D. K. Panda. Is Singularity-Based Container Technology Ready for Running MPI Applications on HPC Clouds? In *Proc. of 10th Intl. Conference on Utility and Cloud Computing (UCC'17)*, pages 151–160. ACM, 2017. 27

[208] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou. A Comparative Study of Containers and Virtual Machines in Big Data Environment. *IEEE International Conference on Cloud Computing, CLOUD*, 2018-July:178–185, 2018. 23, 24

[209] R. Zhang, M. Li, and D. Hildebrand. Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers. In *2015 IEEE International Conference on Cloud Engineering*, pages 365–368, March 2015. 4, 23, 24

[210] N. Zhou, Y. Georgiou, M. Pospieszny, L. Zhong, H. Zhou, C. Niethammer, B. Pejak, O. Marko, and D. Hoppe. Container orchestration on HPC systems through Kubernetes. *Journal of Cloud Computing*, 10, dec 2021. 34

[211] I. Zliobaite, A. Bifet, M. Gaber, B. Gabrys, J. Gama, L. Minku, and K. Musial. Next Challenges for Adaptive Learning Systems. *SIGKDD Explor. Newsl.*, 14(1):48–55, Dec. 2012. 32, 106