UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUILHERME REX PRETTO

Janus: A Framework to Boost HPC Applications in the Cloud based on Just-in-Time and SDN/OpenFlow Path Provisioning

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Luciano P. Gaspary

Porto Alegre
April 2021

CIP — CATALOGING-IN-PUBLICATION

Pretto, Guilherme Rex

Janus: A Framework to Boost HPC Applications in the Cloud based on Just-in-Time and SDN/OpenFlow Path Provisioning / Guilherme Rex Pretto. – Porto Alegre: PPGC da UFRGS, 2021.

63 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Luciano P. Gaspary.

1. HPC applications. 2. Cloud Infrastructures. 3. Link Usage-Aware Path Provisioning. 4. Framework. I. Gaspary, Luciano P.. II. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGEMENTS

First, I wish to thank my advisor, Luciano Gaspary, for the opportunity to work with him and the kindness, guidance, and generosity he has given me through this three-year journey. It was a tough time, not only because of a global pandemic but also because of my professional life, and Luciano always supported me during this period. I've had nothing but positive, constructive experiences.

I thank the Federal University of Rio Grande do Sul (UFRGS) and the Institute of Informatics (INF) for the infrastructure they have provided to aid in this research.

Finally, I extend the most generous gratitude to my mother, Rejane, to my sister Gabriela and my future wife Emilia for their love and continual support of my endeavors.

ABSTRACT

Data centers, clusters, and grids have historically supported High-Performance Computing (HPC) applications. Due to the high capital and operational expenditures associated with such infrastructures, we have witnessed consistent efforts to run HPC applications in the cloud in the recent past. The potential advantages of this shift include higher scalability and lower costs. If, on the one hand, app instantiation – through customized Virtual Machines (VMs) – is a well-studied issue, on the other, the network still represents a significant bottleneck. When switching HPC applications to be executed on the cloud, we lose control of where VMs will be positioned and of the paths that will be traversed for processes to communicate with one another. To bridge this gap, we present Janus, a framework for dynamic, just-in-time path provisioning in cloud infrastructures. By leveraging emerging software-defined networking principles, the framework allows for an HPC application, once deployed, to have interprocess communication paths configured upon usage based on least-used network links (instead of resorting to shortest, pre-computed paths). Janus is fully configurable to cope with different operating parameters and communication strategies, providing a rich ecosystem for application execution speed up. Through an extensive experimental evaluation, we provide evidence that the proposed framework can lead to significant gains regarding runtime. Moreover, we show what one can expect in terms of system overheads, providing essential insights on how better benefiting from Janus.

Keywords: HPC applications. Cloud Infrastructures. Link Usage-Aware Path Provisioning. Framework.

RESUMO

Data centers, clusters e *grid* têm historicamente suporte para aplicações de computação de alto desempenho (HPC). Devido aos altos gastos de capital e operacionais associados a essas infraestruturas, presenciamos esforços consistentes para executar aplicações HPC na nuvem, recentemente. As vantagens potenciais dessa mudança incluem maior escalabilidade e baixos custos de manutenção. Se, por um lado, a instanciação de aplicações - por meio de máquinas virtuais (VMs) personalizadas - é um problema muito estudado, por outro, a rede ainda representa um gargalo significativo. Ao alternar as aplicações HPC para serem executados na nuvem, perdemos o controle de onde as VMs serão posicionadas e dos caminhos que serão percorridos para que os processos se comuniquem entre si. Para preencher essa lacuna, apresentamos Janus, uma estrutura para provisionamento de caminho dinâmico e *just-in-time* em infraestruturas de nuvem. Aproveitando os princípios de rede definidos por software emergentes, a estrutura permite que uma aplicação HPC, uma vez inicializada, tenha caminhos de comunicação entre processos configurados com base na utilização dos *links* de rede menos congestionados (em vez de recorrer a caminhos pré-computados mais curtos). Janus é totalmente configurável para lidar com diferentes parâmetros operacionais e estratégias de comunicação, fornecendo um rico ecossistema para acelerar a execução das aplicações. Por meio de uma extensa avaliação experimental, fornecemos evidências de que o *framework* proposto pode levar a ganhos significativos em relação ao tempo de execução. Além disso, mostramos o que se pode esperar em termos de sobrecarga do sistema, fornecendo *insights* essenciais sobre como obter melhor proveito do Janus.

Palavras-chave: Aplicações HPC, Infraestruturas de Nuvem, Provisionamento de caminho baseado em uso de link, Framework.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BFS	Breadth First Search
BCMS	Bounded Congestion Multicast Scheduling
BT	Block Tri-Diagonal
CG	Conjugate Gradient
DCN	Datacenter Networks
DFS	Depth-First Search
ECMP	Equal-Cost MultiPath
EP	Embarrassingly Parallel
ETC	Ethernet Technology Consortium
FEC	Forward Error Correction
GbE	Gigabit Ethernet
HDR	High Dynamic Range
HPC	High-Performance Computing
LUAR	Link Usage-Aware Routing
LU	Lower-Upper Gauss-Seidel
IaaS	Infrastructure as a Service
IBA	InfiniBand Architecture
IBTA	InfiniBand^{SM} Trade Association
IPoIB	IP over InfiniBand
IT	Information Technology
MPI	Message Passing Interface
MG	Multi-Grid
NIST	National Institute of Standards and Technology

NPB	NAS Parallel Benchmarks
OSPF	Open Shortest Path First
OVSDB	Open vSwitch Database Management Protocol
PCS	Physical Coding Sublayer
PaaS	Platform as a Service
PSSR	Port-Switching Based Source Routing
PoD	Point of Delivery
QoS	Quality of Service
RAM	Random-Access Memory
SaaS	Software as a Service
SP	Scalar Penta-Diagonal
SSH	Secure Shell
SDN	Software-Defined Network
TCAM	Ternary Content-Addressable Memory
ToR	Top-of-Rack
VM	Virtual Machine

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Handling massive amounts of data is commonplace for most modern scientific, engineering, and business applications. As these applications need to target big data-related challenges, while delivering expected results promptly, they frequently pose large computing power requirements. In this context, High-Performance Computing (HPC) becomes a key factor for speeding up data processing. To this end, HPC solutions have traditionally taken advantage of cluster, grid and data center infrastructures for running applications having those computing power requirements (GUPTA et al., 2016). More recently, we have witnessed consistent efforts to run HPC applications in the cloud. The pay-per-use cost model makes cloud computing a promising environment for HPC, which can be provided with instant availability and flexible scaling of resources (*i.e.*, elasticity).

Although the allocation of virtual machines in the cloud (for HPC application execution) has been extensively studied, the inter-process communication rates still represent the main performance bottleneck (LI; ZHANG; LUO, 2017). In addition to the possibility of virtual machines to be deployed physically far from each other into the cloud, the provisioned paths are typically static and based on shortest paths (potentially traversing congested, high delay links). The dynamic nature of communication patterns observed in most HPC applications makes these limitations even more prominent. In summary, subjecting HPC flows to high network latencies is *still* one of the leading open research challenges that cloud environments have to cope with to be able to offer a suitable infrastructure for HPC.

Previous studies (EVANGELINOS; HILL, 2008; GUPTA et al., 2016; NETTO et al., 2018b; ROLOFF et al., 2017; WALKER, 2008; WITTE et al., 2020; KOTAS; NAUGHTON; IMAM, 2018) have assessed the feasibility of using public clouds for HPC. Their findings suggest that clouds were not designed for running tightly coupled HPC applications. The main limitation is the poor network performance resulting from I/O virtualization overhead, processor sharing, and usage of commodity interconnection technologies. Lee *et al.* (LEE et al., 2016) and Faizian *et al.* (FAIZIAN et al., 2017) have identified limitations of network technologies used by cloud infrastructures, namely the use of simplistic routing schemes, which may result in degraded communication performance. To overcome this limitation, they have proposed routing mechanisms that avoid congested paths, which however are computed priorly to the execution of an HPC application and therefore can cope neither with network traffic fluctuation nor with varying

communication patterns.

In this thesis, we proposed Janus, a framework for dynamic, just-in-time path provisioning in cloud infrastructures aiming to speedup HPC applications. It systematically (re)program paths to avoid congested links and reduce runtime, given an application's current communication patterns. To this end, we devised an approach that continuously monitors network link conditions to find the least used path between nodes. In addition to the framework, another equally important contribution is the formalization of a *Link Usage-Aware Routing* (LUAR) strategy, which plays a crucial role in reducing end-to-end communication delays. We also contributed by providing: (*i*) a detailed discussion on the state-of-the-art literature about the usage of SDN as an enabler of cloud infrastructures to execute HPC applications; (*ii*) a description of our prototypical proof-of-concept implementation; and (*iii*) a detailed experimental evaluation of LUAR's efficacy and efficiency. In addition to confirming LUAR's benefits over traditional path provisioning, we extensively discuss different aspects that influence higher speed up gains (providing insights for users and researchers interested in further improving the proposed approach).

The remainder of the thesis is organized as follows. In chapter II, we cover the foundations for building our proposal and some of the most prominent related work. In chapter III, we describe the overarching conceptual solution and its main components. In chapter IV, we explain Janus proof-of-concept implementation. In chapter V, we present the evaluation and discuss the obtained results. In chapter VI, we conclude the thesis with some final remarks and prospective directions for future research.

2 BACKGROUND AND STATE OF THE ART

In this chapter, we provide the fundamentals of cloud computing, discussing the main definitions of the term, the different types of service classifications, and their forms of availability. Network solutions for cloud computing will also be presented, focusing on topologies, routing strategies, and technologies. Afterwards, we review the fundamentals of SDN and OpenFlow, controllers, and OpenFlow switches. Finally, we present the most prominent approaches using SDN to design new routing/forwarding strategies in the context of HPC.

2.1 Cloud Computing

Cloud computing may be considered one of the largest transformations in technology, heavily adopted by all kinds of users. In the early to mid-2010s, as companies started moving their workloads off-premise, it has changed the nature of how organizations work. Today, the area is evolved; serverless computing provides the ability to outsource software code to automatically run remotely and at scale, with resource efficiency and cost reduction. Connectivity to the cloud is now ubiquitous, to the point that almost any business can connect its existing infrastructure to the cloud, being able to run any type of application.

The fundamental idea of cloud computing is the access and management of information regardless of place or platform. In this way, cloud computing makes it possible to use data storage and computing power, which can be shared through a computational infrastructure and quickly escalated due to new demands, with minimal management effort or interaction of the service provider. In 2011, the *National Institute of Standards and Technology* (NIST) defined several cloud computing characteristics (MELL; GRANCE et al., 2011). It is composed of five essential characteristics, three service models, and four deployment models, briefly described next. Additionally, we describe a few modern deployment models.

Essential Characteristics. Cloud computing has five essential characteristics: on-demand self-service, broad network access, resource pooling, rapid elasticity or expansion, and measured service. The *on-demand self-service* means that a consumer must be able to automatically allocate computational resources without human interaction, such as server time and network storage. Implementing user self-service allows customers to access the services they want quickly. This is one of the most attractive cloud features

as users get the resources they need very quick and easy, as opposed to what happens in traditional environments, in which requests often take days to be fulfilled, causing delays in projects and initiatives.

Broad network access is another feature, where capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (*e.g.,* mobile phones, tablets, laptops, and workstations). This ubiquitous availability of services means that resources are available in any part of the world as long as there is an Internet connection. For example, an executive based in Brazil can perform his/her roles during business travel, accessing his/her company's online resources hosted in the United States via the Internet connection in China.

The third characteristic is *resource pooling*. The service provider's computational resources are organized to serve multiple consumers, with both physical and virtual resources dynamically arranged according to consumer demand. There must be a sense of location independence. So, the consumer does not have exact control of where the resources used are located. Still, it must be possible to specify that location at a high level of abstraction (e.g., country, federation unit, or data center).

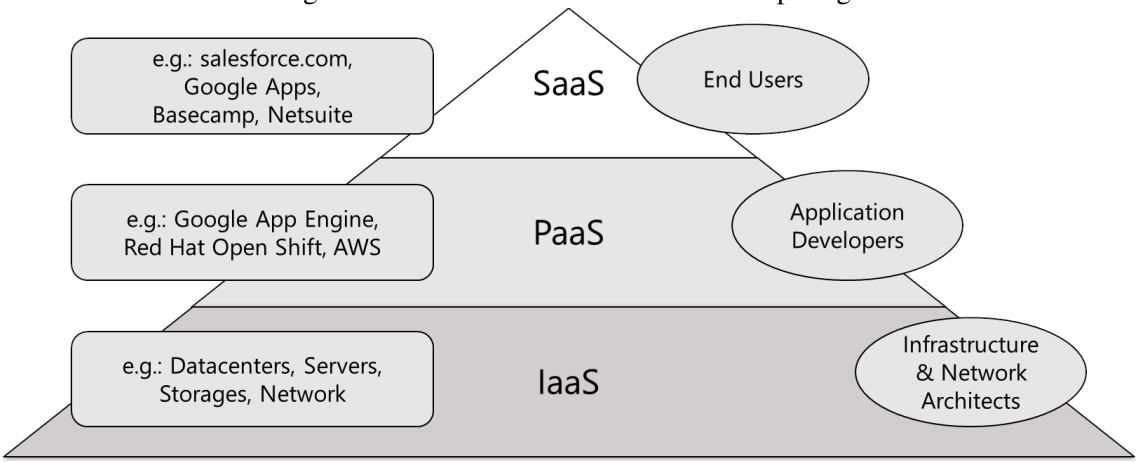
Rapid elasticity is the ability to elastically allocate and release resources. To the consumer, the capabilities available for provisioning often appear unlimited and can be appropriated in any quantity at any time. So, a research lab executing a few HPC applications will be able to handle an unexpected load of hundreds of HPC applications without worrying about the need to purchase new servers to support these seasonal executions.

The last essential characteristic is *measured service*. Cloud systems provide mechanisms to measure resource usage through an appropriate metric system. These measurements are paramount to control and optimize cloud resources automatically. Typical measurements are data storage, computing power processing, bandwidth, and active user accounts. These measurements can also be reported, providing the provider and consumer transparency to the total utilized service. For example, in this work, we use such measurement capabilities to make better decisions regarding the network paths to be taken by packets (*i.e.*, less congested ones) exchanged by HPC application processes.

Service Models. NIST (MELL; GRANCE et al., 2011) also defines three service models for cloud computing, as one can observe in Figure 2.1. The first of these is *Software as a Service* (SaaS), sometimes referred to as on-demand software, which can provide consumers with the use of applications in the cloud infrastructure. These applications are accessible to many customers through an interface such as a web browser or a

specific program. The consumer has no control over the cloud infrastructure management (servers, operating systems, storage, network, or individual application capabilities, except limited application configuration options). Examples of this type of model are business applications such as accounting, management, enterprise resource planning, content, and service management software. This model has been extensively incorporated into the strategy of the largest software companies on the market.

Figure 2.1 – Service models for cloud computing.



Platform as a Service (PaaS) defines the ability to provide the deployment in the cloud infrastructure of content developed by the customer himself or applications created using programming languages, libraries, services, and tools supported by the provider. The consumer still has no control over the management of the cloud infrastructure (servers, storage, operating systems, or network) but has control over how the application is implemented and, possibly, the application's hosting environment's configuration settings. PaaS offerings make it easy to deploy applications without the cost and complexity of acquiring and managing the underlying hardware, software, and hosting provisioning features, providing the facilities needed to support the full lifecycle of building and delivering web applications, in addition to services available entirely from the Internet.

Finally, the definition of *Infrastructure as a Service* (IaaS) is the ability to provide consumers with the possibility of handling processing, storage, network, and other fundamental computing resources. The infrastructure is delivered to consumers on demand while being fully managed by the service provider. Consumers now have control over all the operating system's functionalities, storage, and application distribution. They may also have limited access to network components, for example, firewalls. As we will present and describe ahead, our proposed solution best fits this particular service model.

It provides the operator with mechanisms to choose an appropriate packet forwarding strategy to speed up HPC applications.

Deployment Models. In addition to the service models, NIST (MELL; GRANCE et al., 2011) also defined models for the deployment of clouds: *public cloud*, *private cloud*, *community cloud*, and *hybrid cloud*. *Public clouds* have their infrastructure available to the general public, making it possible for any user to know where the service is located and access it. A public cloud can be owned, managed, and operated by a company, an educational institution, or a government organization. Conversely, in *private cloud* models, the cloud infrastructure is used exclusively by an organization, which can be a local or remote cloud, and it can be managed by the company itself or by third parties. This model adopts policies for access to services. The techniques used to provide such features can be at the level of network management, service provider configurations, and authentication and authorization.

Another interesting model is the *community cloud*. In this deployment model, several organizations share a cloud with common interests and have similar security, policy, and flexibility requirements. This type of model can exist locally or remotely and is usually administered by a company in the community or third parties. An example of this is the clouds maintained by government public services.

The *hybrid cloud* model comprises two or more distinct clouds – private, community, or public – which remain as individual entities but are interconnected by a standardized or proprietary technology that enables data and application portability. This model offers the benefits of multiple deployment models and allows one cloud to extend either capacity or a cloud service's capability by aggregation, integration, or customization with another cloud service. A use case for this type of model is one where IT organizations use public cloud computing resources to meet seasonal capacity needs that can not be delivered by the private cloud.

In the last few years, other models have been discussed, such as *Big Data cloud* and *HPC cloud*. Initially, the need to transfer large amounts of data to the cloud, and data security, hampered the cloud adoption for big data. However, as data originates in the cloud and with the advent of bare-metal servers, the cloud has become a solution for use cases including geospatial analysis and business analytics (YANG et al., 2017). *HPC cloud* refers to the use of cloud computing infrastructure and services to execute high-performance computing (HPC) applications (NETTO et al., 2018a). These applications need considerable computing power and memory and are traditionally executed on clus-

ters. However, since 2016 a couple of companies started to provide cloud infrastructure capable of running such applications (*e.g.,* R-HPC, Amazon Web Services, Gomput, and Penguin Computing).

2.2 Network Solutions for Cloud Computing

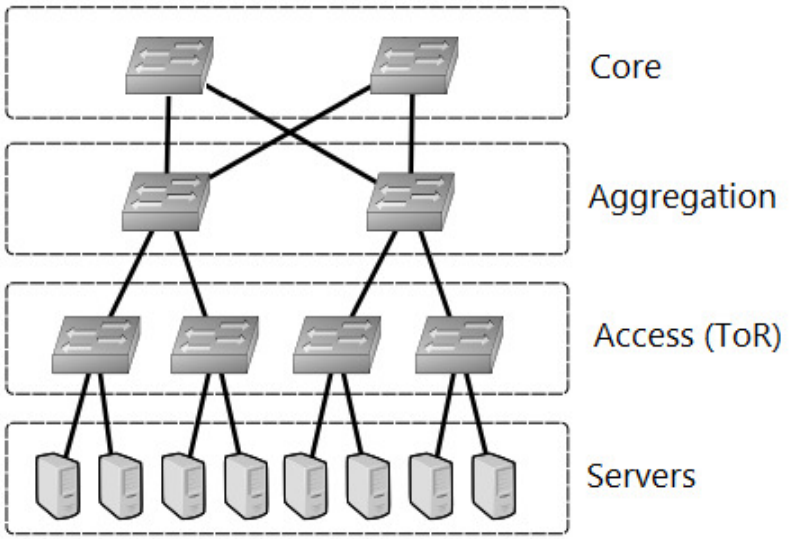
Cloud Computing performance depends heavily on networking. Any limitations or failures of the networking infrastructure (e.g., inside and between data center domains) can seriously impair the support of data-intensive and high-performance cloud applications (MOURA; HUTCHISON, 2016). Consequently, the deployment of Cloud Computing solutions in distributed data centers, concurrently with the universal users' access to the Internet, is challenging the research and standardization communities to modify existing network functionalities. The need for these network changes is fueled by emerging Cloud Computing usage scenarios with dynamic load, data mobility, addressing/routing based on data alternatively to IP destination, heterogeneous resources, federation, and energy-efficiency. This section describes the topologies, routing strategies, and technologies that make cloud computing possible.

Topologies. In the following paragraphs, we present an overview of the main topologies used on data centers. The topology is a representation of how switches, routers, and servers are connected. Typically, a topology is a set of forwarding devices and servers represented by a node, and the links connecting nodes are the edges.

Figure 2.2 shows a canonical three-tiered multi-rooted tree-like physical topology. Data centers such as Oktopus (BALLANI et al., 2011) and used by universities (BENSON; AKELLA; MALTZ, 2010) have been implementing this topology. The Core is composed of routers that interconnect switches in the Aggregation layer. The Aggregation layer has devices connecting the Core and Top-of-Rack (ToR) components. The Access layer is composed of ToR switches connecting servers mounted on every rack. In this type of topology, the components are organized to increase redundancy. Each ToR switch is typically connected to multiple Aggregation switches and every Aggregation switch to various Core switches. A three-tiered network is commonly implemented in data centers with more than 8,000 servers (AL-FARES; LOUKISSAS; VAHDAT, 2008).

Providers employ this topology to reduce costs and increase resource utilization, helping them achieve economies of scale. However, it has many drawbacks, such as limited bisection bandwidth constrains server-to-server capacity [(CURTIS et al., 2012),

Figure 2.2 – A canonical three-tiered tree-like datacenter network topology.



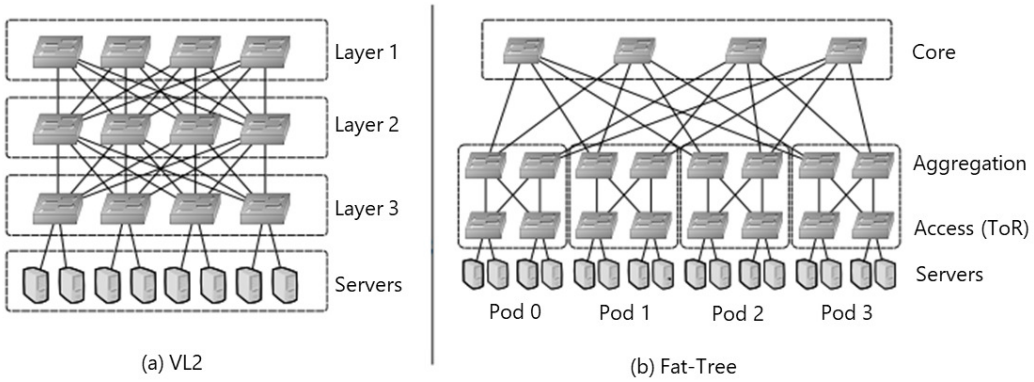
(CURTIS; KESHAV; LOPEZ-ORTIZ, 2010)], and multiple paths are poorly exploited. For example, just one single path is used within a layer-2 domain by spanning tree protocol. Novel network architectures have been proposed to solve these limitations; they can be organized into three classes (POPA et al., 2010): switch-oriented, hybrid switch/server, and server only topologies.

Switch-oriented topologies use commodity switches to perform routing functions and use a clos-based design. A Clos network (CLOS, 1953) has multiple layers of switches, where each switch in a given layer is fully connected to all switches in the upper and lower layers providing path diversity and low bandwidth degradation in case of failures. Two distinct proposals follow the Clos design (Figure 2.3): VL2 (GREENBERG et al., 2009) and Fat-Tree (AL-FARES; LOUKISSAS; VAHDAT, 2008). VL2 provides multiple uniform paths between servers and is used in large-scale data centers. Fat-Tree, in turn, is organized in a tree-like structure. These architectures offer a high capacity. The downside is the increase in wiring costs caused by the number of links.

Hybrid switch/server topologies shift complexity from network devices to servers. This means that servers perform routing while fixed numbers of hosts are interconnected through mini-switches. One benefit of this approach is the innovation allowed as hosts are more straightforwardly customizable than commodity switches. Other benefits are fault-tolerance and richer connectivity. Figure 2.4 shows DCell (GUO et al., 2008) and BCube (GUO et al., 2009), which are two topology examples and can arguably scale up to millions of servers.

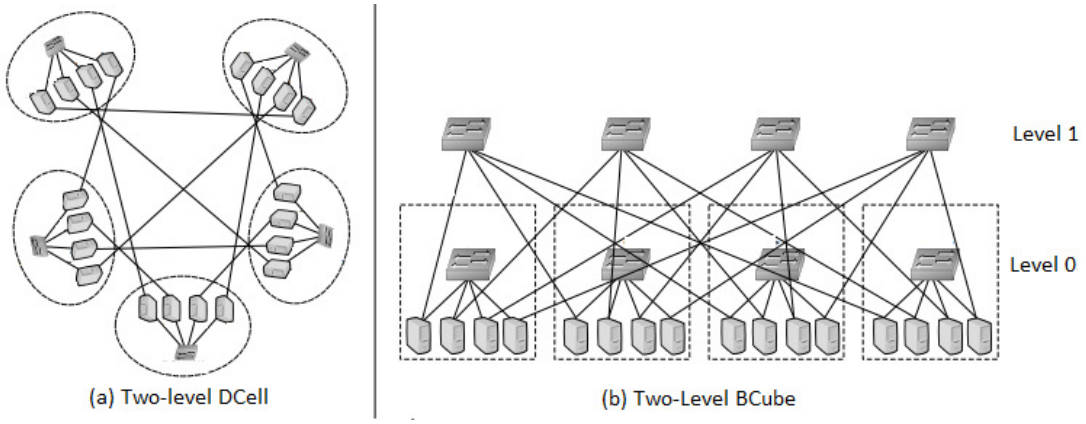
DCell is a recursively-built structure that forms a fully-connected graph using only

Figure 2.3 – Clos-based topologies.



commodity switches (instead of high-end switches of traditional DCNs (datacenter networks)). Similarly, BCube is a recursively-built structure that is easy to design and upgrade. Additionally, BCube provides low latency, and in case of link or switch failure, it has low degradation of bandwidth. BCube clusters (a set of servers interconnected by a switch) are interconnected by commodity switches in a hypercube-based topology.

Figure 2.4 – Hybrid switch/server topologies.

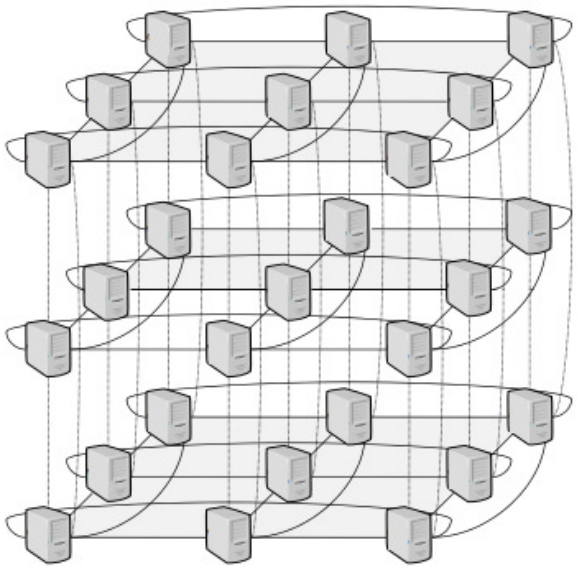


Notwithstanding the benefits, DCell and BCube require many NIC ports at end-hosts, causing overhead at servers and increasing wiring costs. In particular, DCell results in multiple non-uniform paths between hosts, and typically level-0 links are more utilized than other links (creating bottlenecks). Opposed to that, BCube provides multiple uniform paths but uses more switches and links than DCell (GUO et al., 2008).

Server only topology is composed exclusively of servers that perform all network functions. CamCube (ABU-LIBDEH et al., 2010) is an example of such architecture. It is inspired by Content Addressable Network (CAN) (RATNASAMY et al., 2001) overlays and uses a 3D Torus topology with k servers along each axis. Figure 2.5 shows a 3-ary CamCube topology with a total of 27 servers. Some positive aspects of this topology are the following: i) it provides robust fault-tolerance guarantees, ii) improves innovation

with key-based server-to-server routing, and iii) allows each application to define specific routing techniques.

Figure 2.5 – Example of 3-ary CamCube topology.



As one can observe, there are effective options for data center topologies, and they differ regarding operation goals. In general, providers are profit-driven: they choose the topology focusing on the lowest cost, even if it cannot achieve all properties desired for a data center network running heterogeneous applications from distinct tenants.

Routing Strategies. Data center networks have different requirements when compared to traditional enterprise networks. Instead of a handful of paths between hosts and predictable communication patterns observed on conventional enterprise networks, DCNs require many available paths to achieve horizontal scaling of hosts with unpredictable traffic matrices [(AL-FARES; LOUKISSAS; VAHDAT, 2008) and (GREENBERG et al., 2009)]. Due to this requirement, data center topologies usually present path diversity. This means that it is possible to connect servers (hosts) through multiple paths in the network. Furthermore, many cloud applications (*e.g.*, simple web search or complex MapReduce) require substantial bandwidth (AL-FARES et al., 2010). Therefore, routing protocols play an essential role in enabling the network to deliver high bandwidth by exploring all possible topology paths. In fact, our work explores how to use such path diversity to maximize the usage of all possible links between hosts (and minimize the chances to select congested ones). Next, we present different routing approaches.

Equal-Cost MultiPath (ECMP) (HOPPS et al., 2000) aims to utilize all possible paths with the same cost through uniformly spreading traffic among paths using flow

hashing. The routing protocol calculates these paths. However, the static flow-to-path mapping implemented by ECMP does not take flow size and network utilization into account (RADHAKRISHNAN et al., 2013). As a downside, this may result in degrading overall network performance (AL-FARES et al., 2010).

Hedera allows dynamic flow scheduling for general multi-rooted trees with extensive path diversity. Hedera seeks to maximize network utilization with a low scheduling overhead of active flows. This approach uses a central OpenFlow controller (MCKEOWN et al., 2008) with a network global view making it possible to obtain flow statistics, compute new routes, and install the new paths on the devices. With the information collected from switches, Hedera considers the flow-to-path mapping as an optimization problem and uses a simulated annealing metaheuristic to efficiently look for feasible solutions close to the optimal one in the search space.

Port-Switching based Source Routing (PSSR) (GUO et al., 2010) is proposed for the SecondNet architecture with arbitrary topologies and commodity switches. This approach uses source routing. Thus, every node in the network needs to know the complete path to reach a given destination. As the data center is administered by a single entity (*i.e.*, the intra-cloud topology is known in advance), it represents the path as a sequence of output ports in switches. Then, it is stored in the packet header. As servers may have multiple neighbors connected via a single physical port (e.g., in DCell and BCube topologies), PSSR introduces the use of virtual ports.

Bounded Congestion Multicast Scheduling (BCMS) (GUO; DUAN; YANG, 2013) intends to achieve bounded congestion and high network utilization in Fat-trees. It can reduce traffic by using multicast, thus minimizing performance interference and increasing application throughput (LI et al., 2013). BCMS utilizes OpenFlow to collect incoming flows' bandwidth demands. It then monitors network load, computes routing paths for each flow, and configure switches. Despite the advantages of a dynamic approach, BCMS relies on a centralized controller. So, this solution might not scale in environments under highly dynamic traffic patterns such as the cloud.

Similar to BCMS, Code-Oriented explicit multicast (COXcast) (JIA, 2013) also focuses on routing application flows through the use of multicasting techniques, trying to improve network resource sharing and reducing traffic. COXcast uses source routing, so all information regarding destinations are added to the packet header. More specifically, the forwarding information is encoded into an identifier in the packet header and, at each network device, is resolved into an output port bitmap by a node-specific key.

COXcast can support many multicast groups, but it adds some overhead since all routing information is stored in the packet.

Interconnection Technologies. Data center networks are complex environments that need to employ technologies to guarantee low communication latency and high throughput requirements. Amongst existing layer-2 technologies, Infiniband Architecture (IBA) and Gigabit Ethernet are the most used interconnect families used on supercomputers listed on the top500 (SUPERCOMPUTER, 2020). Each technology gets a portion of 31.0%, and 50.8%, respectively.

Infiniband (PFISTER, 2001) is an industry-standard specification that defines an architecture to interconnect servers, communication infrastructure equipment, storage, and embedded systems. It was developed by the InfiniBand^{SM} Trade Association (IBTA) to provide the levels of availability, performance, reliability, and scalability required for present and future server systems. It provides significantly better performance levels than those that can be achieved with bus-oriented I/O structures. Three fundamental characteristics improve IBA over busses. First, all data transfer is bidirectional point-to-point, not bussed. This avoids arbitration issues, provides fault isolation, and allows scaling to a large size by using switched networks. Each link can support multiple transport services for reliability and multiple virtual communication channels. Second, commands and data are transferred between hosts and devices not as memory operations but as messages. Third, IBA defines a layered hardware protocol (physical, link, network, transport) and a software layer to manage the initialization and the communication between devices.

InfiniBand offers a bandwidth of up to 50 Gbps for a single link at version HDR (High Dynamic Range). TCP/IP communications are mapped to the IBA transport services through IP over Infiniband (IPoIB) drivers provided by the operating system. *Datagram* is the default operational mode of IPoIB, as described in RFC 4391 (CHU; KASHYAP, 2006). The minimum MTU allowed is 2,044 bytes, while the maximum is 4,096 bytes. In turn, the *Connected* mode is described in RFC 4755 (KASHYAP, 2006) and offers a connection-oriented service with a maximum MTU of 2GB. The usage of large MTUs can lead to significant performance gains, especially for large data transfers.

Gigabit Ethernet is the term applied to transmitting Ethernet frames at a rate of a gigabit per second. The most common variant 1000BASE-T is defined on the standard IEEE 802.3ab (IEEE..., 1999). It came into use in 1999 and has replaced Fast Ethernet in wired local networks due to its significant speed improvement over Fast Ethernet and its use of cables and devices that are broadly available, cost-effective, and similar to

preceding standards.

The recently rebranded Ethernet Technology Consortium (ETC), previously known as the 25 Gigabit Ethernet Consortium, announced a new 800 Gigabit Ethernet specification during 2020 and an expanded scope aimed at satisfying the requirements of performance-critical networks necessary for applications in high-performance computing, enterprise datacenters, and machine learning. The move brings the organization beyond its roots in 25 and 50 Gigabit Ethernet and a step closer to the Terabit Ethernet era, propelled by tremendous data demands that dictate ever-faster communications.

ETC made available the 800GBASE-R specification for 800 Gigabit Ethernet (GbE), which implements a new media access control (MAC) and Physical Coding Sublayer (PCS). The specification is based on two sets of existing 400 GbE logic from the IEEE 802.3bs pattern (giving a total of 32 x 25-Gbps PCS lanes), redefined so data can be distributed across eight 106 Gbps physical lanes. By reusing the PCS, standard forward error correction (FEC) is achieved, supporting compatibility with existing physical layer specifications.

2.3 Software-Defined Networking

Although the SDN paradigm was conceived relatively recently, the idea of programmable networks was imagined much longer ago, around the mid-1990s (CALVERT, 2006). The growing ubiquity of the Internet and the growth of a more diverse array of applications led to the necessity of developing and testing new protocols in realistic network scenarios. Motivated by this need, researchers began to investigate ways to open up network control and make networks more easily programmable.

Active networking was the first significant novel approach to network programmability. The fundamental concept behind it was the presence of a network programming interface. This interface enabled developers to access resources on individual network nodes and program specific functions on them. Through this programmability, active networking aimed to keep the network core simple and lower innovation barriers.

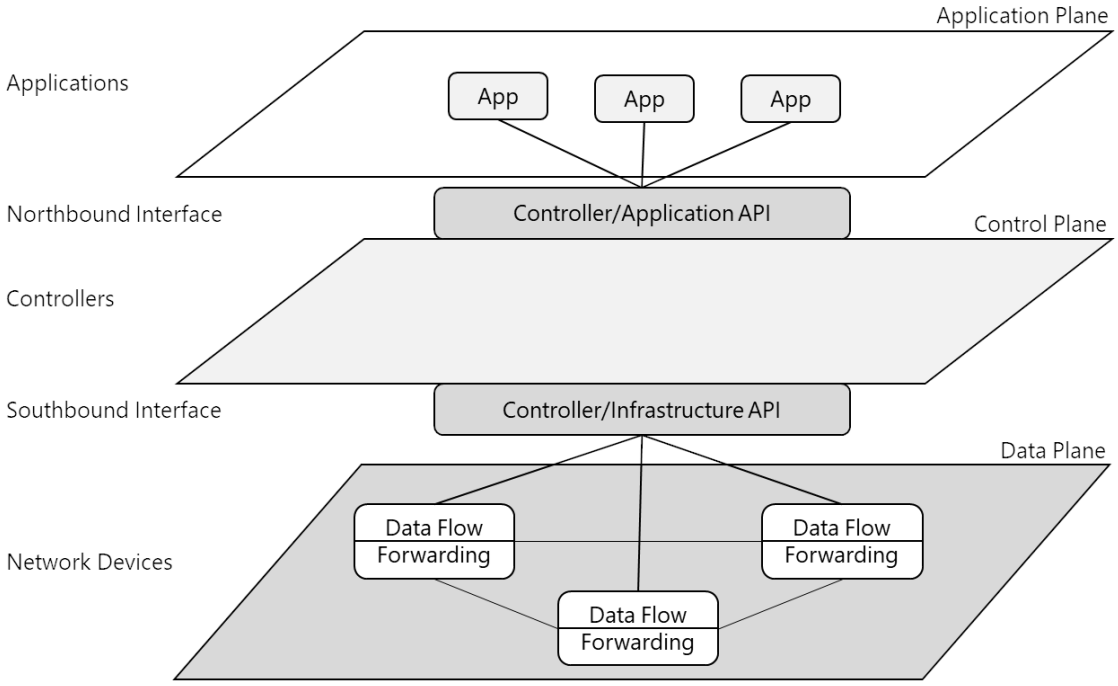
Eventually, active networking failed to reach widespread adoption. Feamster et al. (FEAMSTER; REXFORD; ZEGURA, 2014) argue that this failure may have been due to “the lack of an immediately compelling problem or a clear path to deployment”. Despite this, active networking was a milestone for subsequent efforts to achieve network programmability and related concepts such as network virtualization.

Fundamentals of SDN and OpenFlow. Software-Defined Networking laid out on the concept of active networking but trying to solve a narrower and more streamlined set of problems. The main idea was to split the planes to address routing and configuration management issues. On the one hand, the control plane is responsible for making routing decisions based on network policies. On the other hand, the data plane follows the control plane decisions and simply forwards traffic. It enables centralized network intelligence while abstracting low-level network infrastructure information.

Figure 2.6 presents an overview of the components present in an SDN architecture. The application plane consists of applications implementing services provided to users/devices through the network. These applications interact with the SDN controller through APIs (the Northbound Interface). The ideal scenario for the Northbound Interface is that a standard is established, allowing independence on the programming language and the controller used. However, this has not yet been defined, and each controller specifies its own API.

The control plane concentrates the "intelligence" of the software-defined network. It offers a centralized control logically (which does not necessarily imply the existence of a single physical controller). This layer provides abstractions, essential services, and APIs for developers. This means that a developer no longer needs to know all the details of packet transmission to define a network policy, facilitating its development, and decreasing the chance of errors.

Figure 2.6 – Overview of the OpenFlow architecture.



The Southbound Interface is the bridge between the control and data planes. This interface is also an API. Through it, SDN controllers can communicate the application requirements to the network, reprogramming the equipment to perform numerous functions, such as flow control, firewall, intrusion detection systems (IDS), and routing. This reprogramming is carried out by adding or removing rules from the flow tables, described in detail ahead. All communication that passes through this layer uses a secure communication channel.

The OpenFlow protocol is the most notable and widely adopted implementation of the SDN paradigm. It specifies an open and well-defined communication protocol between network devices and the control plane, independent of the equipment vendor. Moreover, it requires network devices to fulfill specific requisites (*e.g.*, set of operations they must be able to perform on incoming packets). The OpenFlow protocol has continuously developed since conceived in 2008 (MCKEOWN et al., 2008), with at least five major versions released since then.

OpenFlow networks may optionally employ multiple controllers simultaneously to improve reliability, allowing the switch to continue to operate in OpenFlow mode if a controller or controller connection fails. In this case, while the control plane remains logically centralized, it is physically distributed among several controllers. These controllers coordinate the management of the switch amongst themselves to help synchronize controller handoffs.

OpenFlow Controllers. The controller has a global view of the whole network, including data of the topology, interfaces, statistics, and events. Network administrators can use this information to develop high-level algorithmic solutions for network management, enabling dynamic, automated network control applications. These applications communicate directly with the controller, taking advantage of the high-level information provided to make better network management decisions. The controller configures forwarding devices in the network, ensuring that they behave as defined by the application.

Switch-controller communication happens following a well-defined protocol. Through the utilization of this protocol, the controller can send messages to interact with forwarding devices. The most common messages are related to requesting the addition, removal, and modification of flow rules. It is also possible to request network statistics, such as device ports and data flow information, which may be crucial for decision-making.

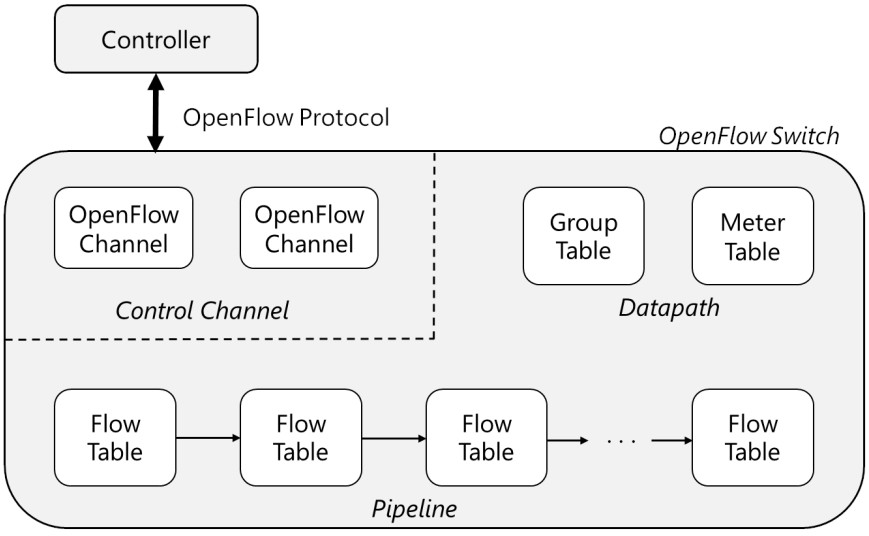
The OpenFlow Logical Switch and OpenFlow controllers are connected by the interface OpenFlow channel. Through this interface, the controller configures and manages

the switch. It also receives events from the switch and sends packets out to the switch. The control channel of the switch may support only a single OpenFlow channel with a single controller or may support multiple OpenFlow channels enabling various controllers to share the management of the forwarding device. Independent of the setup, all OpenFlow channel messages must be formatted according to the OpenFlow switch protocol.

Taking into consideration the standardized switch-controller communication protocol, multiple OpenFlow controllers implementations can be found. These implementations allows developers to use different programming languages such as Python (*e.g.*, POX and Ryu) or Java (*e.g.*, Beacon and Floodlight). Also, it is possible to encounter implementations providing different set of features such as enabling the federation of multiple controllers (*e.g.*, Flowvisor (SHERWOOD et al., 2009)) or providing capabilities for network virtualization (*e.g.*, (NASCIMENTO et al., 2011) and OpenVirteX (AL-SHABIBI et al., 2014)).

As one of the widely used controllers, Ryu is an open SDN controller designed to increase the network's agility by making it simple to both manage and adapt how traffic is handled. The Ryu controller provides software components with well-defined application programming interfaces (API), making it easy for developers to create new network management and control applications. Examples of APIs provided by Ryu are: Open vSwitch Database Management Protocol (OVSDB), NETCONF, XFlow (Netflow and Sflow) and other third-party protocols. This component based approach helps organizations customize deployments to meet their specific needs; developers can quickly and easily modify existing components or implement their own to ensure the underlying network can meet their applications' changing demands.

Figure 2.7 – Main components of an OpenFlow switch.



OpenFlow Switches. Unlike traditional network paradigms, SDN-enabled network devices are limited to two simple operations, processing and data forwarding. An incoming packet is treated depending on how the controller programs a forwarding device. OpenFlow employs the concept of "traffic flows" to classify and treat network packets, allowing varying degrees of granularity. Each flow has some matching fields and at least one action. Matching fields distinguish among different network flows and typically represent packet header fields, such as source and destination addresses and transport ports. Actions define the operations that can be performed on packets depending on the described match flow. In addition to forwarding packets through specific network interfaces on a switch, actions can be used to manipulate packet header fields (*e.g.*, changing the source or destination IP). Tables 2.1 and 2.2 lists all matching fields and actions available in the latest publicly available version of the OpenFlow protocol (1.5.1).

Figure 2.7 depicts the main elements within each OpenFlow device. Switches store into the flow tables the information regarding flow matching fields and actions. When a new packet is received, it is processed by a flow table; the packet is matched against flow entries of the flow table to select a flow entry. If a match is found, the device executes the operations defined in the "actions" field. If a table-miss happens, the behavior on a table miss depends on the table configuration. It may communicate with the controller to determine which action should be performed. Actions specified in flow tables may direct packets for further processing in a different flow table or group actions (stored in the group table). OpenFlow switches also track individual and aggregate statistics regarding traffic that traversed them. The meter table is used to measure packet rates and implement QoS (Quality of Service)-related operations such as rate-limiting.

Table 2.3 contains the basic set of OpenFlow stat fields. Each flow table of the switch must support these fields. Optional stat fields must be included in the stats structure, but they need to be supported by the flow table. This set of fields enables measuring traffic on the network, building algorithms to take advantage of such information. Janus is one example that uses these measures to understand what device interfaces are underutilized to route new data to them.

Flow tables are often distributed between two different types of memory, namely TCAM (Ternary Content-Addressable Memory) and RAM (Random-Access Memory). On the one hand, TCAMs are highly suited for flow table operations, as they allow parallel lookups with very high performance. However, it comes with a price, as they are 400x more expensive (LIAO, 2012) and consume 100x more power (SPITZNAGEL; TAYLOR;

Table 2.1 – OpenFlow matching fields.

Field	Description
IN_PORT	Switch input port.
IN_PHY_PORT	Switch physical input port.
METADATA	Metadata passed between tables.
ETH_DST	Ethernet destination address.
ETH_SRC	Ethernet source address.
ETH_TYPE	Ethernet frame type.
VLAN_VID	VLAN id.
VLAN_PCP	VLAN priority.
IP_DSCP	IP DSCP (6 bits in ToS field).
IP_ECN	IP ECN (2 bits in ToS field).
IP_PROTO	IP protocol.
IPV4_SRC	IPv4 source address.
IPV4_DST	IPv4 destination address.
TCP_SRC	TCP source port.
TCP_DST	TCP destination port.
UDP_SRC	UDP source port.
UDP_DST	UDP destination port.
SCTP_SRC	SCTP source port.
SCTP_DST	SCTP destination port.
ICMPV4_TYPE	ICMP type.
ICMPV4_CODE	ICMP code.
ARP_OP	ARP opcode.
ARP_SPA	ARP source IPv4 address.
ARP_TPA	ARP target IPv4 address.
ARP_SHA	ARP source hardware address.
ARP_THA	ARP target hardware address.
IPV6_SRC	IPv6 source address.
IPV6_DST	IPv6 destination address.
IPV6_FLABEL	IPv6 Flow Label
ICMPV6_TYPE	ICMPv6 type.
ICMPV6_CODE	ICMPv6 code.
IPV6_ND_TARGET	Target address for ND.
IPV6_ND_SLL	Source link-layer for ND.
IPV6_ND_TLL	Target link-layer for ND.
MPLS_LABEL	MPLS label.
MPLS_TC	MPLS TC.
MPLS_BOS	MPLS BoS bit.
PBB_ISID	PBB I-SID.
TUNNEL_ID	Logical Port Metadata.
IPV6_EXTHDR	IPv6 Extension Header pseudo-field
PBB_UCA	PBB UCA header field.
TCP_FLAGS	TCP flags.
ACTSET_OUTPUT	Output port from action set metadata.
PACKET_TYPE	Packet type value.

TURNER, 2003) per megabit than RAM. On the other hand, RAM offers limited performance as lookups must be performed sequentially. Therefore, the TCAM area tends to be

Table 2.2 – OpenFlow actions.

Field	Description
OUTPUT	Output to switch port.
COPY_TTL_OUT	Copy TTL "outwards" – from next-to-outermost to outermost
COPY_TTL_IN	Copy TTL "inwards" – from outermost to next-to-outermost
SET_MPLS_TTL	MPLS TTL
DEC_MPLS_TTL	Decrement MPLS TTL
PUSH_VLAN	Push a new VLAN tag
POP_VLAN	Pop the outer VLAN tag
PUSH_MPLS	Push a new MPLS tag
POP_MPLS	Pop the outer MPLS tag
SET_QUEUE	Set queue id when outputting to a port
GROUP	Apply group.
SET_NW_TTL	IP TTL.
DEC_NW_TTL	Decrement IP TTL.
SET_FIELD	Set a header field using OXM TLV format.
PUSH_PBB	Push a new PBB service tag (I-TAG)
POP_PBB	Pop the outer PBB service tag (I-TAG)
COPY_FIELD	Copy value between header and register.
METER	Apply meter (rate limiter)

Table 2.3 – OpenFlow stats.

Field	Description
DURATION	Time flow entry has been alive.
IDLE_TIME	Time flow entry has been idle.
FLOW_COUNT	Number of aggregated flow entries.
PACKET_COUNT	Number of packets matched by a flow entry.
BYTE_COUNT	Number of bytes matched by a flow entry.

extremely limited in OpenFlow switches, adding to the necessity of adequately managing flow tables and keeping track of table occupation.

After presenting an overview of Software-Defined Networking, we now proceed to a review of the related work. This review includes a discussion on the employment of SDN as a platform to enable efficient communication, especially in the context of HPC.

2.4 Related Work

The Software-Defined Network (SDN) paradigm has been providing the means to design new packet routing/forwarding strategies to diverse types of computer networks (*e.g.*, internet service providers, corporate, and data center networks). SDN is a promising enabling paradigm in this context for two main reasons. First, it decouples the control and data planes, shifting network decisions to a centralized entity. Second, it provides a network's global view, making it easier to devise innovative control mechanisms. We start

this section with a brief overview of how SDN has been explored to design novel routing/forwarding mechanisms (for different purposes). After this introduction, we review existing strategies to optimize cloud networks for HPC, including how SDN is boosting new methods for efficient data transmission in this field.

2.4.1 SDN as Enabler of New Routing Schemes.

Bera *et al.* (BERA; MISRA; OBAIDAT, 2019) propose Mobi-Flow, a solution designed to maximize the overall performance of a software-defined access network. It resorts to dynamic SDN path reprogramming as end-users move around a determined environment. The authors show that the proposed scheme, based on a greedy approach, helps minimize the associated costs in data delivery by nearly 40% compared to Open Shortest Path First (OSPF). Guillen *et al.* (GUILLEN et al., 2018) propose a solution to efficiently use the network infrastructure in the context of a distributed storage system by employing an SDN-based multipath routing mechanism. Preliminary results show that, by applying the proposal, the overall throughput increases by almost four times in "long flows" and around 30% in the case of "short flows" while resource usage remains balanced. As another illustrative example, Guan *et al.* (GUAN et al., 2018) introduce an overlay approach seeking to improve transmission quality for Internet applications requiring stringent QoS guarantees.

2.4.2 Optimizing the Cloud Network for HPC.

Several feasibility studies about using public clouds for HPC point out the network communication overheads as a significant barrier. Other investigations propose strategies to reduce these overheads. Two classes of solutions stand out: those resorting to topologies that attempt to maximize connectivity between nodes and those that explore resource and/or network management. Given the nature of our proposed approach, next, we revisit work on the general problem of executing HPC applications in the cloud and recent proposals that apply varying management tactics for efficient inter-process communication paths.

Lightweight Virtualization. Gupta *et al.* (GUPTA et al., 2016) investigate network bottlenecks in commercial cloud infrastructures and propose new mechanisms to

improve the performance experienced by HPC applications. Essentially, they employ lightweight virtualization and grant Virtual Machines (VMs) native access to physical network interfaces. Despite the overhead reduction in average turnaround time (up to 2x) and throughput (up to 6x), the authors recognize that their proposal may not be enough to improve the performance of HPC applications. Similarly, Ramakrishnan *et al.* (RAMAKRISHNAN et al., 2012) claim that virtualization of network resources accounts for the main performance issue, being responsible for at least 60% delay and throughput degradation if compared with typical local cluster and supercomputer options.

Efficient Packet Scheduling. Tokmakov *et al.* (TOKMAKOV et al., 2019) introduce a novel traffic management algorithm that combines Rate-limited Strict Priority and Deficit Round-Robin for latency-aware (and fair) scheduling. It was designed to improve link utilization and prioritize ultra-low latency flows (*e.g.*, of HPC and edge computing applications) in Ethernet-based setups. However, round delays can negatively result from the algorithm as queues are served in a Round-Robin fashion. Similarly, Hauser *et al.* (HAUSER; PALANIVEL, 2017) propose a dynamic network scheduler for cloud data centers. Based on polled network statistics, it orchestrates bandwidth limitation mechanisms to prevent network congestion and unfair bandwidth utilization.

Custom SDN-based InfiniBand/Dragonfly Networks. Mauch *et al.* (MAUCH; KUNZE; HILLENBRAND, 2013) introduce an approach to use InfiniBand in a private virtualized environment. It allows for an individual network configuration to operate with QoS mechanisms. Similarly, Alsmadi *et al.* (Alsmadi; Khamaiseh; Xu, 2016) propose a method for logically slicing the network and isolating applications from one another. While the former is based on a complex architecture that requires interfacing with several APIs and protocols to configure the network, the latter cannot handle path reallocation in case of a switch or link failure. Moreover, both proposals deploy static paths that cannot change during application execution.

Motivated by the fact that InfiniBand is increasingly available in commercial clouds, Zahid *et al.* (Zahid et al., 2018) implement an InfiniBand setup that, based on a feedback control and optimization loop, can adjust its operating parameters. It enables an HPC network to dynamically adapt to varying traffic patterns, resource availability, and workload distribution while conforming to service provider-defined policies. Lee *et al.* (LEE et al., 2016) address one of InfiniBand central weakness for HPC, namely the employment of simple routing schemes such as *destination-mod-k* routing. The authors propose an enhancement to the technology, incorporating OpenFlow-style SDN capability to provi-

sion uncongested paths. While the proposal is a cornerstone towards the employment of programmable networks to speed up HPC applications in the cloud, it fails by requiring routes to be determined before the execution of an application. It, therefore, wastes a valuable opportunity to adapt the network to meet fluctuating application communication demands.

Along the same lines as the work by Lee *et al.*, Faizian *et al.* (FAIZIAN et al., 2017) include SDN-related functionality to Dragonfly networks and propose a set of routing schemes to achieve the performance required by HPC applications. One limitation of this work is the lack of a mechanism for accurate traffic demand estimation, making it difficult to make optimal global decisions. Another limitation is that even when several paths are available, only two (*i.e.*, a primary and a backup) are considered for forwarding a packet. A final shortcoming is that these paths are provisioned upon application instantiation and cannot change.

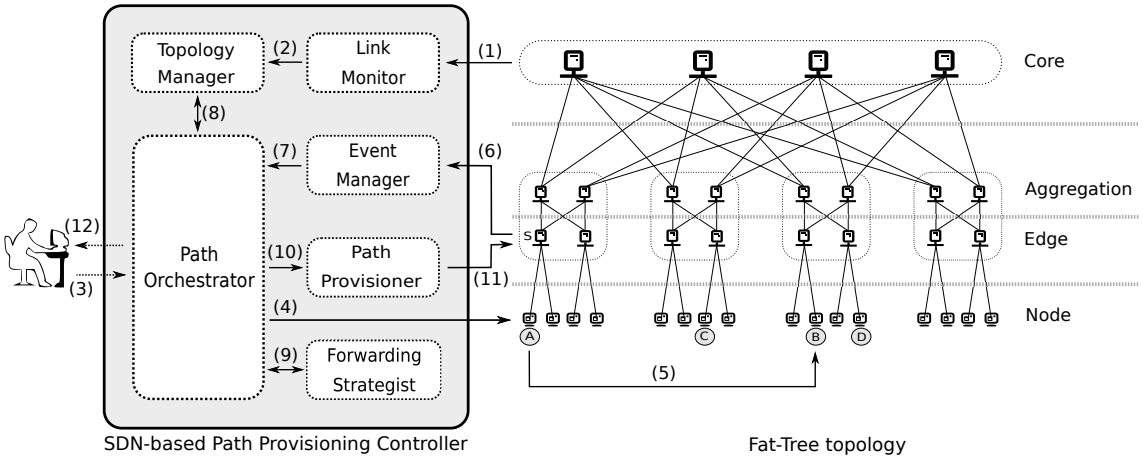
Summary. SDN has been investigated for different purposes on network management. Specifically for HPC scenarios, recent studies employ lightweight virtualization, latency-sensitive flow prioritization, and path provisioning to increase the efficiency of inter-process communications. Although a few proposals advance in path provisioning methods, they still do not take full advantage of the functionality provided by software-defined networks and are static. This thesis proposes Janus, a framework that contributes to dynamically selecting and provisioning the least used paths during application runtime.

3 THE PROPOSED PATH PROVISIONING APPROACH

As a relevant research step further in comparison with the related work, we propose a framework based on SDN for *just-in-time path selection and provisioning* in cloud infrastructures. Janus continuously monitors the network conditions and, given the current communication patterns of an HPC application, systematically (re)programs paths to avoid congested links and reduce end-to-end delays. The proposed approach is *complementary* to the VM placement and instantiation process, which we assume will be taken care of by a third party resource management system such as OpenStack (SEFRAOUI; AISSAOUI; ELEULDJ, 2012) or OpenNebula (MILOJIČIĆ; LLORENTE; MONTERO, 2011).

Figure 3.1 introduces the basis of our solution, highlighting its main conceptual components and their interactions. In the following sections, we describe them in detail, starting with (*i*) the SDN-based Path Provisioning Controller, then (*ii*) illustrating a working example, and finally (*iii*) presenting the accompanying proposed routing strategy.

Figure 3.1 – Components of the proposed solution and their interactions.



3.1 SDN-based Path Provisioning Controller

The SDN-based Path Provisioning Controller consists of six components: *Link Monitor*, *Topology Manager*, *Event Manager*, *Path Orchestrator*, *Forwarding Strategist*, and *Path Provisioner*. Next we summarize the role of each of these components.

Link Monitor. This component plays a vital role in periodically gathering traffic statistics from the forwarding devices. Assuming the infrastructure is OpenFlow-enabled, this information can be obtained via `OFPPortStatsRequest` requests. The infras-

tructure operator must tune the frequency of this process. Highly frequent requests favor accuracy while sparsed ones favor low intrusiveness. After each round of information collection, the Link Monitor calculates the amount of traffic that traversed each link and derives its average utilization. This information is calculated according to Equation 3.1 and sent to the Topology Manager component.

$$LkUsage_i(t, t-1) = LkCapacity_i - \frac{tx\_bytes(t) - tx\_bytes(t-1)}{\Delta(t, t-1)} \tag{3.1}$$

Topology Manager. It maintains updated information about the topology of the cloud network infrastructure in the form of an *annotated graph*. Forwarding devices (*e.g.*, OpenFlow switches) are abstracted as vertices while physical links are represented as edges. The average utilization of each link, measured by the Link Monitor, is used to denote the edge weight. The reader should keep in mind that (*i*) the representation of the topology might be updated in response to changes in the infrastructure (*e.g.*, due to a link failure – event `OFPErrorMsg`) and (*ii*) the weights are updated regularly in response to periodic link usage measurements carried out by the Link Monitor.

Event Manager. Whenever a new flow starts between two processes of an HPC application or a previously existing one remained idle for a long time, there will be no path provisioned in the network to route its packets. The forwarding device that is closer to the source host then triggers an event (`OFPPacketIn`) that is received by the Event Manager component. This component intermediates communication with the Path Orchestrator for a path selection procedure.

Path Orchestrator. The orchestrator, as the name implies, coordinates the process of path selection and provisioning. The component reacts to both changes in the physical topology (informed by the Topology Manager) and events of the type `OFPPacketIn` (reported by the Event Manager). It consumes (updated) information made available by these components, invokes the Forwarding Strategist for the determination of a path and resorts to the Path Provisioner for its deployment. The Path Orchestrator is also responsible for interfacing with the human network operator (*e.g*, to receive configuration commands and send him/her notifications about the path provisioning process).

Forwarding Strategist. The Forwarding Strategist can implement different approaches for determining a path between a source and a destination (HPC application) process. It uses an updated view of the annotated graph and location information of the processes to be "interconnected". The problem addressed by this component is then of a *graph-based path finding* one. As a proof of concept, in this thesis, we propose a *Link*

Usage-Aware Routing strategy, which favors shortest and uncongested paths. It is described in detail in Section 3.3.

Path Provisioner. This component is responsible for performing the actual deployment of end-to-end paths on the network infrastructure. This process unfolds in the invocation of a coordinated set of `OFPFlowMod` commands to install forwarding rules on the devices that (are calculated to) compose the paths. The duration of these rules is determined by the operator, who can favor either path freshness/efficiency or stability/reuse.

3.2 A Working Example

To illustrate how the described components interact, we introduce now a working example. Initially, recall that the Link Monitor is expected to be continuously monitoring the traffic to consolidate statistics from the cloud network infrastructure (Flow 1 in Figure 3.1). The gathered information (*tx_bytes*) is sent to and stored by the Topology Manager (Flow 2).

The process of deploying an HPC application consists of (*i*) instantiating and configuring the VMs (with any off-the-shelf resource management system), as well as (*ii*) requesting our proposed SDN-based Path Provisioning Controller (via the Path Orchestrator component) to manage the path provisioning process (Flow 3). Should the third-party resource management system expose an API that can be used to control the resource provisioning life-cycle, then it is possible to integrate (*i*) and (*ii*) to have the whole application deployment/execution process dealt with by the SDN-based Path Provisioning Controller (3 and 4).

Consider the situation in which an HPC application has been deployed and that the process running on node A needs to communicate with the process on node B (5). When the first packet from A reaches its closest forwarding device (*s* in the figure), the device will not know how to route it further and will send an `OFPPacketIn` event to the Event Manager (6). This component will then pass the event to the Path Orchestrator (7).

At this point, the Path Orchestrator consults the Topology Manager to receive updated information about the current topology and the utilization of the links (8). Next, it invokes the Forwarding Strategist to determine an optimized route to connect processes running on A and B (9). Finally, it requests the Path Provisioner (10) to deploy the forwarding rules on the network devices (11). From this moment on, the packet that triggered the event is reintroduced by the controller into the network, and the remaining

flow packets will be transmitted through the configured route.

The process above is executed for each new application communication flow (or when the physical topology changes). To avoid frequent communications with (and interventions from) the external controller, the operator can resort to two system parametrization options. The first is setting up a higher value for the `idle_timeout` associated with the flow rules configured on the forwarding devices. The consequence will be that provisioned paths will be kept active for more extended periods (assuming that short inter-process *no* communication periods will be succeeded by new message exchanges). The second option is *proactively* provisioning paths for which there exists a single possible route, or that will be undoubtedly necessary along the execution of a whole application (*e.g.*, master/worker communication channels). An in-depth analysis of the additional performance gains (*i.e.*, in application runtime) reached with these configuration options are out of the scope of this work and are left as future work.

3.3 Link Usage-Aware Routing (LUAR)

In this section, we propose and formalize our strategy – called *Link Usage-Aware Routing* (LUAR) – designed to efficiently compute the best path from a source to a destination point in a cloud network topology. LUAR is one of the possible approaches to be implemented by the Forwarding Strategist. It considers the best path to be the shortest, least utilized, currently available path. In the future, we plan to investigate other algorithms and approaches to this task.

Algorithm 1 shows the pseudo-code of LUAR. The algorithm has five input parameters: a set V of nodes in the topology graph; a vector of adjacency lists E, where $E(v)$ lists the nodes connected to node v; a link utilization matrix U, where $U(v, u)$ indicates the utilization of the link that connects node v to u; and the source and destination nodes s and t. The algorithm can be logically divided into two blocks. Next, we detail each block.

The first block (Lines 1–12) is an adaptation of the Breadth-First Search (BFS) algorithm extended to record, for every node, its possible predecessors in any of the shortest paths starting from the source. The algorithm maintains two main data structures in Block 1, Predecessors and Seen, that indicate, respectively, the possible predecessors for each node v and the first level in which they were seen during the search. These structures are initialized with nil values for every node in the graph (Line 2). In Line 3, we

Algorithm 1 Compute the shortest, least utilized, currently available path from source s to destination t.

Require: V, E, U, s, t

1: ▷ ***Block 1***: Generate predecessor tree for the graph starting from node s.
2: Predecessors$(v) \leftarrow nil, \forall v \in V$**;** Seen$(v) \leftarrow nil, \forall v \in V$
3: Predecessors$(s) \leftarrow$ EMPTYARRAYLIST()**;** Seen$(s) \leftarrow 0$
4: $N \leftarrow$ QUEUE$([s])$
5: **while** $N \neq \varnothing$ **do**
6: $v \leftarrow$ POP(N)
7: **for** $u \in E(v)$ **do**
8: **if** Seen$(u) = nil$ **then**
9: Predecessors$(u) \leftarrow$ ARRAYLIST$([v])$
10: PUSH(N, u)
11: Seen$(u) \leftarrow$ Seen$(v) + 1$
12: **else if** Seen$(u) =$ Seen$(v) + 1$ **then** APPEND(Predecessors$(u), v)$
13: ▷ ***Block 2:*** Compose the shortest paths and find the least utilized one.
14: $W \leftarrow$ARRAYLIST$([(t, 0, 0)])$**;** $\rho_{best} \leftarrow nil$**;** $au_{best} \leftarrow \infty$
15: **while** LENGTH$(W) > 0$ **do**
16: $v, i, au \leftarrow$LAST(W)
17: **if** $v = s$ **and** $au < au_{min}$ **then**
18: $\rho_{best} \leftarrow$ REVERSED(LIST($\{$FIRSTITEM(w): $w \in W\}$))
19: $au_{best} \leftarrow au$
20: **if** $i <$ LENGTH(Predecessors(v)) **then**
21: APPEND(W, (Predecessors$(v)_i, 0, au + U$(Predecessors$(v)_i, v)$))
22: **else**
23: REMOVELAST(W)
24: **if** LENGTH$(W) > 0$ **then**
25: SECONDITEM(LAST(W)) $\leftarrow$ SECONDITEM(LAST(W)) $+1$

Ensure: ρ_{best}

set Predecessors(s) of source node s to an empty list (since it has no predecessors) and Seen(s) to level 0 (since the search will start from it). The algorithm also maintains an auxiliary variable N, which is a queue of nodes to explore, initialized to contain only the source node s (Line 4).

Lines 5–12 contain the main repeat loop of Block 1. Every iteration of the outermost loop represents the exploration of a level in the graph. Each iteration starts by removing an element from N and assigning it to variable s (Line 6). The algorithm then checks, for each adjacent node $u \in E(v)$ (Line 7), if it has not been visited (Line 8).

In the positive case, the Predecessor(u) is initialized to a list containing only the node currently being explored (*i.e.*, v; Line 9), u is added to the set N of nodes to explore in the next level (Line 10) and is also marked as to have been seen in next level Seen$(v) + 1$ (Line 11). Otherwise, LUAR checks if node u has already been seen in the next level (Line 12), in which case it simply appends v to the list of predecessors of u. This procedure is repeated until all nodes in the graph have been visited, *i.e.*, no item is added to N during an outermost iteration (Line 5). In the end of Block 1, data structure Predecessors has been populated with the predecessors of all nodes in the shortest paths starting from node s.

Block 2 (Lines 13–25) is a modified version of the non-recursive Depth-First Search (DFS) algorithm. The search is started from the destination node t and targets source node s (*i.e.*, a reverse search) and the Predecessors data structure (populated in Block 1) is used instead of the original adjacency list. Along with the search, LUAR records the path being traversed and its aggregate utilization. The path is recorded using an array list W of triples (v, i, au_{cur}), where v is a node, i is index of the next predecessor to explore for v, and au is the current aggregate utilization in the path ending on node v. Block 2 maintains two auxiliary variables (Line 14): the best path ρ_{best} found in the search (initialized as nil); and the aggregate utilization au_{best} of this best path (initialized as ∞).

Lines 15–25 contain the main loop of Block 2. Each iteration starts by loading the last triple from W into variables $v, i,$ and au (Line 16). Line 17 checks whether the current node v is the source node s and the current aggregate path utilization au is smaller than the best known value au_{best}. In case both conditions are true, LUAR builds the current path by creating a list from the nodes listed in W, which are the first items of each triple (Line 18), and updates the best known path ρ_{best} to be it. Line 19 also updates the minimum known aggregate utilization au_{best} to be the current one au. Regardless of the previous conditions, if there is predecessors that should still be explored for node v (Line 20), a new triple is added to W containing the next predecessor of v to be explored and adding up the utilization from the predecessor to v to the current aggregate utilization (Lines 21). Otherwise, LUAR removes the last triple from W, and if W does not become empty, it increments by one the index of the next predecessor to explore from the last triple. This process is repeated until all of the shortest paths from source node s to destination node t have been analyzed. At the end of the algorithm, variable ρ_{best} is returned and indicates the shortest, least utilized, currently available path.

The worst-case complexity of LUAR is given by $O(n+m)$, where n is the number of nodes and m refers to the number of links in the topology, since both of the adapted algorithms (BFS and DFS) have complexity $O(n + m)$ and are executed sequentially in our algorithm. Therefore, LUAR runs in polynomial linear time to the number of devices and links in the network and is suited to quickly compute the best path to route traffic from two processes of an HPC application.

4 PROOF-OF-CONCEPT IMPLEMENTATION OF JANUS

Next, we briefly describe our prototype implementation. The presentation is guided by the architectural view depicted in Figure 3.1 and focuses on how the components of the proposed approach are materialized in our implementation using the RYU platform version 4.3 and OpenFlow version 1.3. First, we present the Link Monitor and Topology Manager implementation (Section 4.1). Next, we explain the Forwarding Strategist (Section 4.2). Finally, we detail the implementation of the Path Orchestrator together with the Event Manager and Path Provisioner (Section 4.3).

4.1 Link Monitor and Topology Manager Components

The Link Monitor and Topology Manager mentioned in Section 3.1 are part of a single Python-based procedure deployed on top of RYU at the external controller side and are implemented as two distinct functions. The monitor continuously polls the SDN-based network devices for link traffic statistics. The manager asynchronously collects, processes, and consolidates the statistic events received.

After system initialization, the procedure abstracted in Algorithm 2 is invoked and starts to poll the network. It receives as input an array list with all available forwarding devices in the topology (*sw*), where each sw is an OpenFlow switch, and an *interval* variable determining the period between statistic requests (in seconds). We clarify to the reader that the *interval* parameter is expected to be configured earlier by a human operator (in his/her previous interactions with the Path Provisioner component, whose implementation is explained ahead). While the network is up and running, each sw on the array list is visited (Line 2) every *interval* seconds (Line 5). Lines 3 and 4 generate a PortStatsRequest() to the current switch s, which in OpenFlow is carried out through an `OFPPortStatsRequest` message.

Algorithm 2 Switch Statistic Polling

Require: ARRAYLIST(sw), $interval$

1: **while** True **do**
2: **for** $s \in$ ARRAYLIST(sw) **do**
3: $req \leftarrow PortStatsRequest()$
4: s.SENDMSG(req)
5: Sleep($interval$)

Each switch asynchronously responds with an `EventOFPPortStatsReply` event (to the port statistic request), and it triggers the procedure formalized in Algorithm 3. It has three input parameters. The first is a link utilization matrix U, where $U(v, u)$ indicates the link between the source node (v) and destination node (u). Each matrix cell contains two attributes: the current utilization of the link ($stat$) and the local output port number ($port$) to reach u (from v). The second parameter E is a vector of adjacency lists, where $E(v)$ lists the nodes connected to v. The last parameter is the $PortStatsReplyEvent$, containing the current statistics for each port of a switch. Line 1 parses and temporarily stores in S the switch identification, ports, and statistics for each $Port$ (from the JSON-encoded event). In Lines 2–3, n is assigned the number of $ports$ (contained in S), v is assigned as the source node ($S.id$), and a $Port$ variable is initialized with 0. Each $Port$, parsed from S, is then visited (Line 4), and the corresponding traffic statistic captured by the event is stored in variable $NewStat$ (Line 5). Lines 6–7 search the correct pair of nodes v and u to update the traffic information. The previous link weight is then retrieved from the matrix U and assigned to variable $PrevStat$. Next, function $TrafficAmount$ calculates the exchanged bytes between the current ($NewStat$) and the previous observation ($PrevStat$) and assigns the result in $U(v, u)$ (Line 9). We call the reader's attention that bidirectional links are modeled as two unidirectional links (one in each direction), whose loads are calculated independently.

Algorithm 3 Topology Statistic Consolidation.

Require: $U,\ E,\ PortStatsReplyEvent$

1: $S \leftarrow$ parser($PortStatsReplyEvent$)
2: $n \leftarrow$ PortCount(S)
3: $v \leftarrow S.id$; $Port \leftarrow \{0\}$
4: **while** $Port <= n - 1$ **do**
5: $\quad NewStat \leftarrow S.stat[Port]$
6: $\quad$ **for** $u \in E(v)$ **do**
7: $\quad\quad$ **if** $U(v, u).port == S.port[Port]$ **then**
8: $\quad\quad\quad PrevStat \leftarrow U(v, u).stat$
9: $\quad\quad\quad U(v, u).stat \leftarrow$ TRAFFICAMOUNT($NewStat,\ PrevStat$)
10: $\quad Port \leftarrow Port + 1$

4.2 Forwarding Strategist Component

The Forwarding Strategist is also implemented using Python, and its pseudo-code is illustrated in Algorithm 4. To perform a path selection, the Forwarding Strategist requires as inputs six parameters, namely: V (set of nodes in the topology graph), E, U, source and destination IPs (s, t) where the HPC processes wishing to communicate are located, and selected strategy (*strategy*). The component executes a switch case-like structure (Lines 1-2) to invoke the operator-chosen path computation procedure to run (*i.e.*, LUAR and shortest path in the algorithm). Note that the approach can be extended to support other path selection tactics, in which case the procedure shown needs to be updated. Depending on the strategy invoked, the path computation procedure parameters are different (*e.g.*, Shortest Path does not utilize statistics to select a path, see Lines 3 and 5). However, the outcome is always a path, ρ_{best}, in the format of an array comprised of 3-tuples (hop, in_port, out_port), where hop is a forwarding device in the selected path, in_port is the packet's incoming port, and out_port is the outgoing port.

Algorithm 4 Forwarding strategist.

Require: $V, E, U, s, t, strategy$

1: **switch** $strategy$ **do**
2: **case** $LUAR$
3: $\rho_{best} \leftarrow$ COMPUTELUAR(V, E, U, s, t)
4: **case** $ShortesPath$
5: $\rho_{best} \leftarrow$ SHORTESTPATH(V, E, s, t)
⋮

Ensure: ρ_{best}

4.3 Event Manager, Path Orchestrator, and Path Provisioner Components

Components Event Manager, Path Orchestrator, and Path Provisioner are implemented through some functions in a single Python procedure. Algorithm 5 formalizes the pseudo-code organizing the main actions executed, in a coordinated way, by these components. Block 1 (Lines 2–4) is responsible for handling the receipt of a `Packet_In` event and parsing it. Block 2 (Lines 6–10) coordinates and invokes actions from other components to determine a path. Block 3 (Line 12) ultimately installs the SDN/OpenFlow forwarding rules (to provision the previously determined path) on the network devices.

Path Configuration and Deployment. When the Event Manager receives a new `Packet_In` event, it triggers the execution of Algorithm 5. In addition to the event, the algorithm receives as input two parameters: $idle\_timeout$, responsible for determining the amount of time a path is expected to be kept active (after no further packets for the flow are observed), and $strategy$, informing the approach to be used to select a path. In Block 1, the event is parsed into a structure ev. We use the Ryu packet library to help with this task. Then, the source (s) and destination (t) nodes, *i.e.*, the endpoints of the new path to be provisioned, are picked (from ev). Block two starts by retrieving V, E, and U (described earlier) from the Topology Manager (Lines 6–8). In Line 9, the algorithm invokes the Forwarding Strategist passing as parameters V, E, U, s, t, and $strategy$. It will then wait for the component to return a $Path$. Line 10 translates the high-level representation of the current $Path$ into flow rules to be sent to the switches. Finally, in Block 3, Line 12, the algorithm calls function ADDFLOW with variables $Flow$ and $idle\_timeout$ as parameters. This function will iterate through the OpenFlow switches composing the determined path and, through the execution of the `ofp_flow_mod` primitive, install the new forwarding rules.

Algorithm 5 Path Configuration and Deployment

Require: $PacketInEvent, idle\_timeout, strategy$

1: ▷ ***Block 1***: Receive and parses packet-in event.
2: $ev \leftarrow$ PARSER$(PacketInEvent)$
3: $s \leftarrow ev.SourceNode$
4: $t \leftarrow ev.DestinationNode$
5: ▷ ***Block 2***: Orchestrate components to select the best available path.
6: $V \leftarrow$ RETRIEVENODES$()$
7: $E \leftarrow$ RETRIEVEADJACENTNODES$()$
8: $U \leftarrow$ RETRIEVEUTILIZATIONMATRIX$()$
9: $Path \leftarrow$ FORWARDINGSTRATEGIST$(V, E, U, s, t, strategy)$
10: $Flow \leftarrow$ TRANSLATETONETWORKPRIMITIVES$(Path)$
11: ▷ ***Block 3***: Install rules in the forwarding devices.
12: ADDFLOW$(Flow, idle_timeout)$

General System Configuration and Access. To configure the controller's expected behavior and then execute an application, a human operator must interact with the provisioning controller and carry out the following steps. First, he/she must determine the operation parameters mentioned throughout this section. Second, he/she must specify the hosts that will be available to execute the Message Passing Interface (MPI) applications.

This process can be performed manually or through the previous interaction of our system with a third party resource management system (not prototype in this work). With this setup up and running, it is possible to start the Ryu Controller (and, in the case of our experimental design, also Mininet). Then, to deploy an MPI application, it is possible to access one or more of the available hosts defined, through SSH (Secure Shell), upload and initiate the application.

We highlight that all artifacts described above are available for download and use at GitHub (https://github.com/gpretto/janus-framework).

5 EVALUATION

In this chapter, we evaluate the proposed framework aiming at assessing application execution speedup and costs under varying conditions. We start by detailing the experimental setup. Next, we characterize the workloads employed. Finally, we present and discuss the results obtained with the SDN-based path provisioning controller (with LUAR).

5.1 Experimental Setup

We implemented the cloud infrastructure with Mininet (LANTZ; HELLER; MCKEOWN, 2010). Hosts were instantiated as full-fledged lightweight Linux VM instances. The network infrastructure comprised Open vSwitch version 2.0.2 switches (that support OpenFlow version 1.3) and links of 1 Gbps. The SDN-based Path Provisioning Controller was executed as an application on top of Ryu (RYU, 2019). This experimental environment was deployed on two nodes, each with 2 x 2.0 GHz Intel Xeon E5-2640 v2 Ivy Bridge (Q3'13) processors, with 64 GB DDR3 RAM and running Debian 4.19. One node was responsible for running the end-hosts, while the other executed the SDN Controller.

The right-most side of Figure 3.1 illustrates a simplified version of the cloud network topology used in the experiments. It is a *fat-tree* that, as usual, is organized in three hierarchical layers: *Edge*, *Aggregation*, and *Core*. In our topology, 32 hosts ($h1 \ldots h32$) were attached to 8 edge switches. In the Aggregation layer, we instantiated 8 switches, while in the Core, 4 (totaling 20: $s1 \ldots s20$). The switches in the Edge and Aggregation layer were organized in 4 PoDs (Points of Delivery).

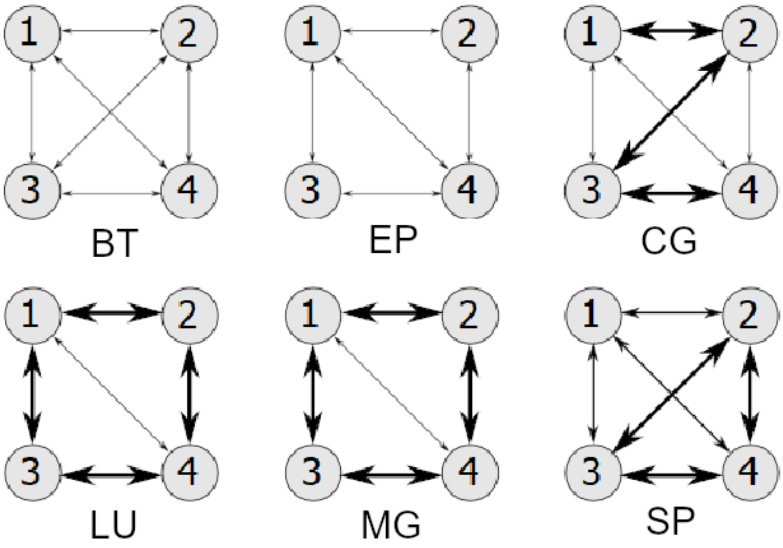
Throughout the evaluation, two distinct scenarios were considered: one named *Sparse*, meaning the VMs were spread across multiple PoDs, and one entitled *Dense*, in which the VMs allocated to execute an HPC application were randomly provisioned within the same PoD. The workloads to which the experiments were submitted are described in the next section (5.2). We compared the results obtained using LUAR with those observed with a more traditional forwarding strategy, namely *Shortest Path*. The primary metric employed for the comparison was *application runtime*. To adequately understand the trade-offs between application efficiency and network programming costs, we examined different idle timeout values for forwarding rules. Each experiment was repeated 30 times. Results have a confidence level of 90%.

5.2 Workloads

Regarding the HPC application used to carry out the experiments, we employed the NAS Parallel Benchmarks (NPB) version 3.3 (BAILEY, 2011). The benchmarks are designed to support the performance evaluation of HPC infrastructures and systems. They are derived from computational fluid dynamics and consist of five kernels and three compact applications. We chose NPB because, in addition to encompassing a wide range of HPC application profiles, it is a reference, extensively employed for high-performance computing evaluation in the literature.

We selected six representative NPB applications, whose communication patterns are illustrated in Figure 5.1 (considering their instantiation with four nodes). Thick edges denote intensive communication demands, while thin ones represent the opposite. Observe that in BT all nodes exchange data with one another, but at low amounts. Similarly to BT, in EP data exchange is also low, but not all nodes communicate directly (*e.g.*, 2 and 3 in the example). CG communications form a string-style pattern, where most of the communications happen in a chain (*e.g.*, node 1 with 2, 2 with 3, 3 with 4). The LU and MG applications take the form of a ring-based topology, with massive information exchange. Finally, SP comprises an initial data distribution step, followed by data-intensive communication between half of the process pairs.

Figure 5.1 – Communication patterns of six NPB applications.



According to recent data center measurements (GUO et al., 2017), a large portion of data center traffic is carried by a small fraction of long-lived, voluminous flows (called *elephant* flows). Similarly, Joy *et al.* (Joy; Nayak, 2015) report that nine out of ten

flows transfer less than 1 MB of data (*mice* flows), while 90% percent of the total bytes transferred belongs to flows larger than 100 MB (*i.e.*, elephant).

In line with the studies mentioned above, we emulated a realistic traffic workload composed of mice and elephant flows. Each experiment had a duration of 15 minutes and was composed of two states: *warmup* ($1 - 3$ minutes) and *steady* ($4 - 15$ minutes) states. During warmup, the number of ongoing background flows increased gradually until reaching the desired level of resource competition. During the steady-state, to maintain the expected overall concurrence level, a new background flow was started whenever an existing flow ended. To create such a new flow, we randomly selected a pair of *producer* and *consumer* processes, and determined its duration to be of up to 4 minutes (respecting the proportion of $9/10$ mice and $1/10$ elephant flows). It was also during the steady-state that we instantiated the HPC application, *i.e.*, one of the NAS benchmarks shown in Figure 5.1, complying with either the Sparse or Dense scenario.

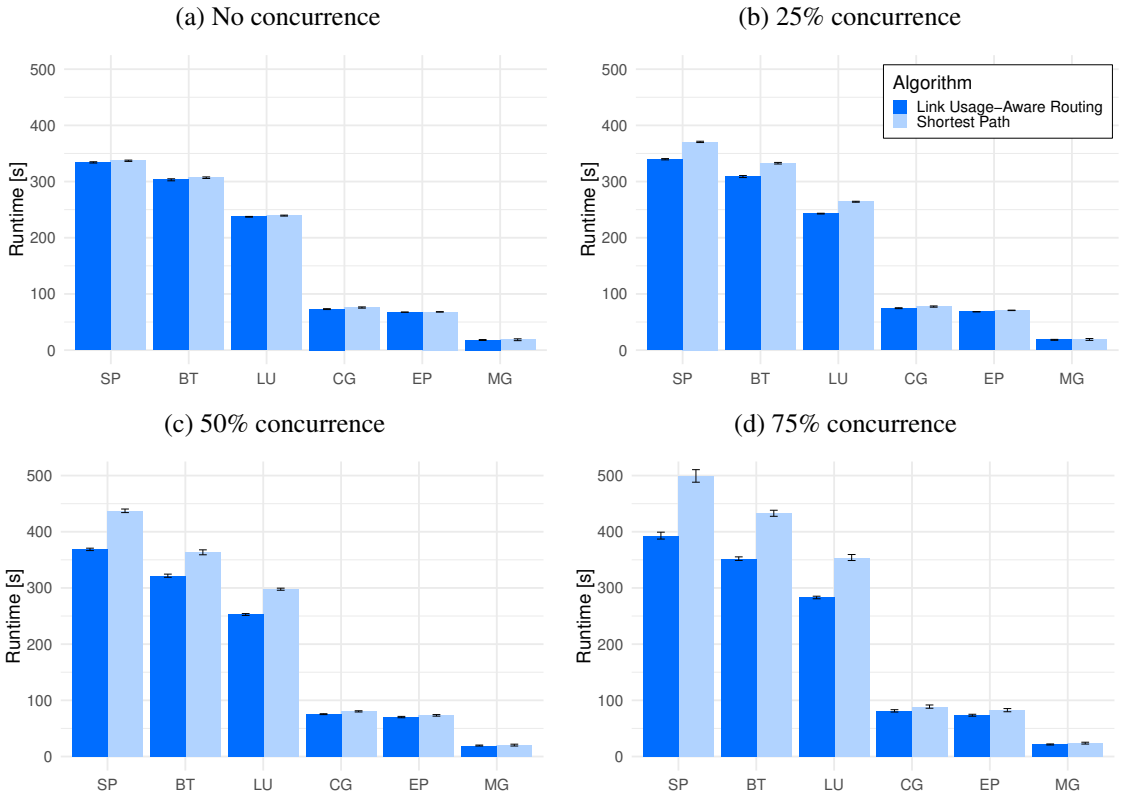
5.3 Results

In this section, we report the results obtained with our mechanism, comparing them with measurements performed using a standard path-provisioning algorithm. Focusing on the primary evaluation metric, *i.e.*, HPC application speed up, we examine how it is impacted by varying controller intervention and flow rule reprogramming frequencies (in an attempt to determine an optimal operation point for the proposed solution).

Application Execution Speed Up. We first present and discuss the results obtained, starting with the Sparse scenario, followed by the Dense one. Figure 5.2 summarizes the application performance when the VMs (and, as a consequence, the HPC application processes) are spread across multiple PoDs. Notice that larger applications tend to take more advantage of our proposed approach. The application runtime observed for BT, LU, and SP is, on average, 41.86 seconds lower (12.71% speedup) than what is achieved with Shortest Path. The gains of LUAR for CG, EP, and MG are more modest, being less than 3-second runtime improvement.

Another important angle from which to analyze the results is the performance gains as a function of the concurrency level. The higher the cloud network's contention, the higher the advantage and value of employing LUAR. On average, it outperforms Shortest Path by 1.65% in (5.2a – no concurrence), 6.12% in (5.2b), 10.70% in (5.2c), and 17.94% in (5.2d – 75% concurrence). Taking both the size of an application and the

Figure 5.2 – Application performance observed for the Sparse scenario.

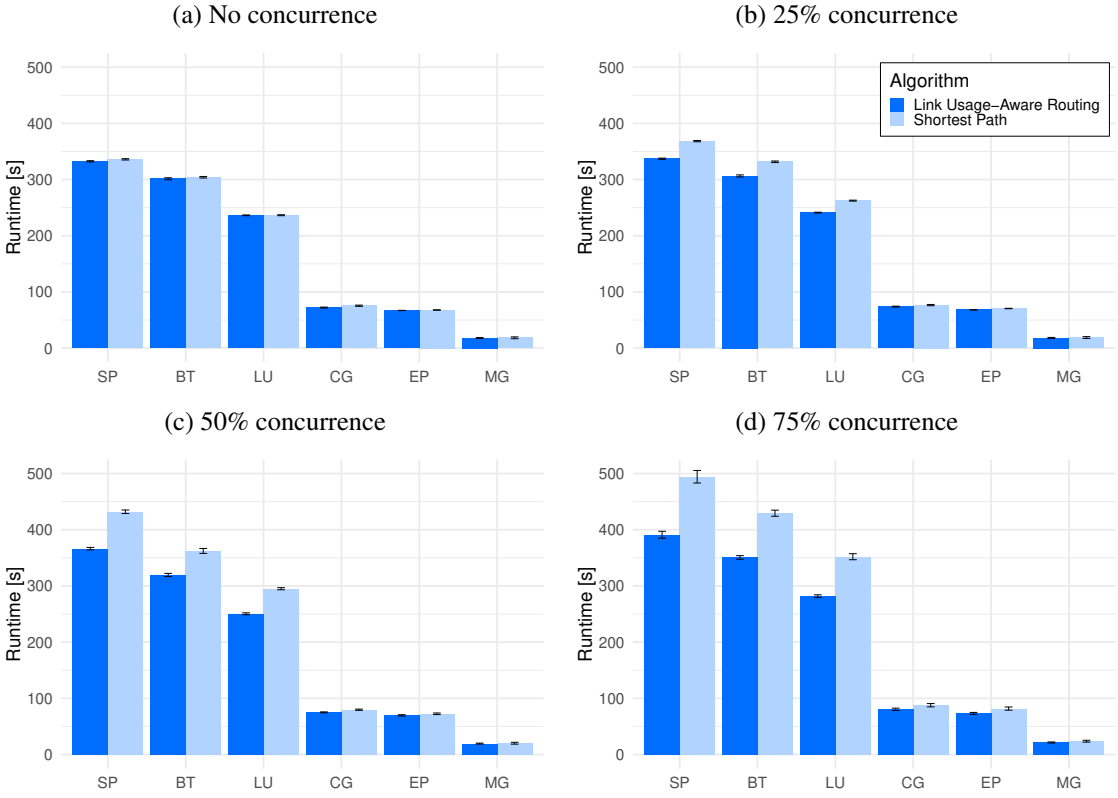


level of concurrence into consideration simultaneously, we observe that the shortest application, MG, exhibits a nearly constant 4.2% runtime improvement with LUAR (against Shortest Path) regardless of concurrency level. Conversely, the largest application, SP, presents consistently increasing gains of 0.86% (5.2a), 9.08% (5.2b), 18.62% (5.2c), and 27.01% (5.2d).

We move now to the scenario in which the VMs (and the HPC application processes) are allocated on hosts of a single PoD. This scenario is expected to be more common but with fewer opportunities to speedup application performance through just-in-time traffic-aware path provisioning. Figure 5.3 shows the results. Despite the reduction of path diversity on the network edge for the Dense scenario, LUAR beats its counterparts. For example, SP executes 17.90% faster than when using Shortest Path under the condition of 50% concurrency. Other applications are too short-lived to take full advantage of LUAR but still run faster. This is the case of CG, which took 79.71 seconds to complete with Shortest Path and 75.09 seconds with LUAR (6.15% faster).

Finally, it is worth analyzing the performance observed comparing both scenarios. Similarly to what is seen for the Sparse setup, Shortest Path performs consistently worse in the Dense scenario, *i.e.*, for all applications and concurrency levels, than LUAR. On

Figure 5.3 – Application performance observed for the Dense scenario.



average, our gains range from 1.68% to 15.52% for the Dense scenario (and from 1.65% to 17.94% for the Sparse one). These results underscore that deciding paths dynamically and with accurate knowledge of network traffic has the potential to allow better routing decisions.

Detailed Quantitative Analysis. We dived into each NAS application's behavior in previous experiments to understand why some applications take better advantage of the mechanism than others. For this analysis, we correlated application speed up, the volume of traffic exchanged between node instances, and communication patterns. Table 5.1 summarizes this information for the Sparse – 75% concurrence scenario, Figure 5.2d. The remaining investigations in this section focus on this scenario only because: (*i*) it is the one where a solution such as LUAR is highly demanded, (*ii*) the results for the other scenarios follow similar trends, and (*iii*) to avoid unnecessary plots.

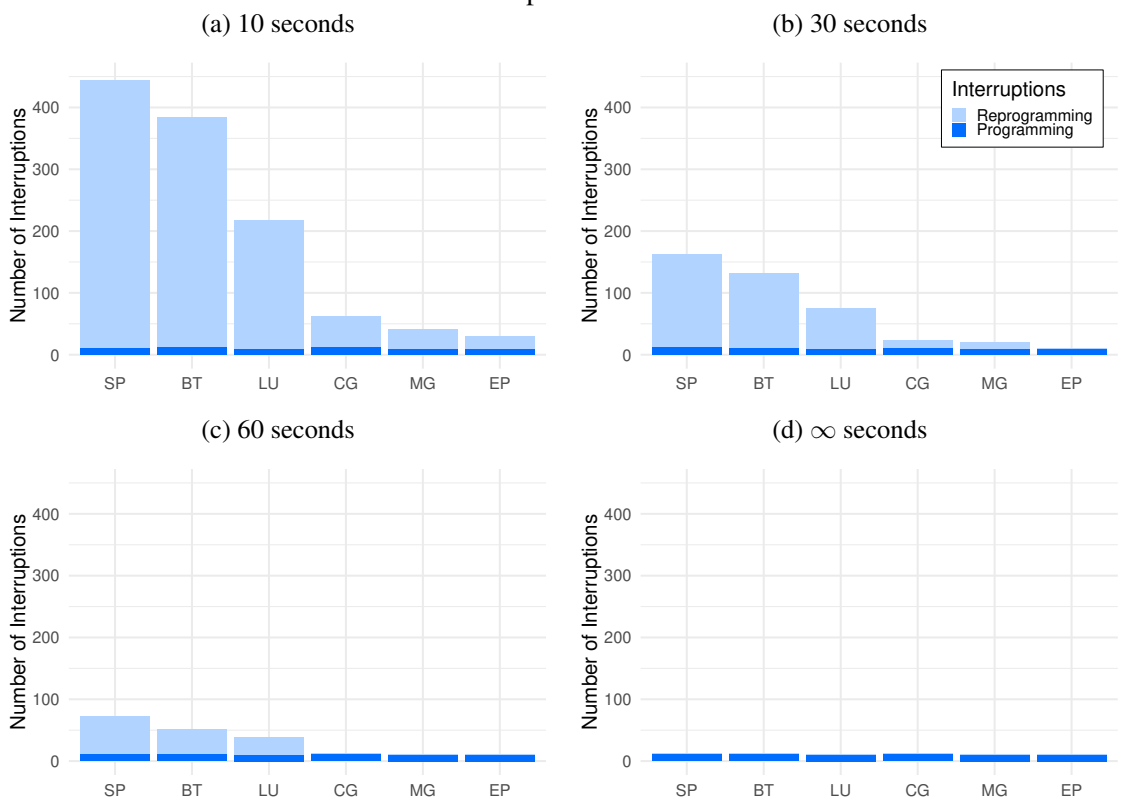
On the one hand, some applications (MG, EP, and CG) execute in less than 90 seconds and exchange less than 150 MB of data. On the other hand, other applications, such as LU, BT, and SP, have runtime higher than 200 seconds and more than 480 MB of data exchanged between nodes. Comparing these two groups, we observe that application speedup for the second group is, on average, of 1.24X using our approach (against

Table 5.1 – Relationship among application speedup, data volume exchanged, and communication patterns for the Sparse – 75% concurrence scenario.

NAS Application	Runtime (s)		Speedup	Volume (MB)	Comm. Pattern
	S-Path	LUAR			
MG	23.82	22.47	1.06X	103.34	Ring
EP	82.51	73.50	1.12X	0.01	All
CG	88.83	81.24	1.09X	137.27	Chain
LU	354.15	283.03	1.25X	491.80	Ring
BT	432.89	352.11	1.22X	976.96	All
SP	499.35	393.13	1.27X	1707.81	All

the Shortest Path). For the first group, characterized by applications that consume fewer network resources, the speedup is, on average, of 1.09X. Thus, one can conclude that network-resource intensive applications, *i.e.*, the ones for which the amount of data exchanged is high, will benefit the most from our approach.

Figure 5.4 – Number of interruptions to the the external SDN controller as a function of varying rule timeout values for the Sparse – 75% concurrence scenario.



Controller Interventions and Execution Delay. We measured the interruptions from the forwarding devices to the external controller during application execution as a function of varying idle timeout values for rule expiration. Figure 5.4 shows the results obtained for the Sparse – 75% concurrence scenario. In the graphs, *programming* interruptions denote the communication with the controller for the first path provision's

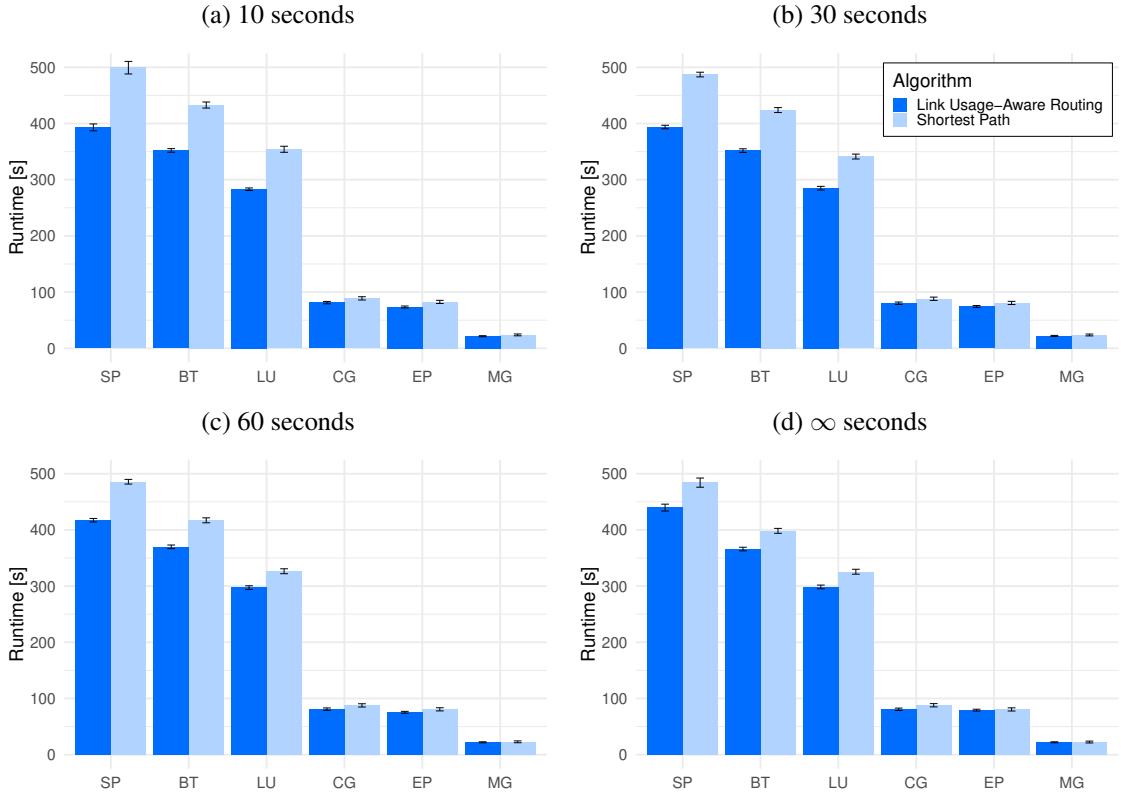
decision/procedure. In turn, *reprogramming* interruptions mean subsequent path provisioning decisions (performed after each timeout and link load reassessment). Note that as the idle timeout value increases, the network's programmability level decreases. The average number of path reprogramming actions considering timeouts of 10, 30, 60 seconds, and non-expiring rules (∞) are, respectively, 185.66, 59.66, 21.33, and 0.

The analysis of the number of interruptions and the results reported in Table 5.1 allows us to suggest that long-duration applications achieve high speedups in part because they do use more extensively the functionality of path reprogramming. More specifically, these applications' processes have a higher probability of using uncongested links to communicate. Conversely, short-lived applications tend to select paths (which may be congested) and do not have time enough to switch to more convenient ones. As an illustrative example from Figure 5.4, scenario 5.4a, while CG redefines forwarding paths 50 times (speedup of 1.09X), SP performs 432 path changes (speedup of 1.27X).

Using the previous results as a starting point for a more in-depth analysis, we also assessed the impact of using different idle timeout values (for forwarding rule expiration) on application runtime. Figure 5.5 shows the results for the Sparse – 75% concurrence scenario. Similarly to the results shown in Figure 5.2, and as a general remark, LUAR exhibits considerable gains against Shortest Path across all scenarios. On average, the former outperforms the latter by 17.94% in (5.5a), 15.02% in (5.5b), 9.71% in (5.5c), and 6.71% in (5.5d). A second important observation from Figure 5.5 is that the shorter the idle timeout value, the faster the execution time of each NAS application. For example, LU executes in 299.54 seconds in (5.5d), and gradually decreases its execution time to 297.30 seconds in (5.5c), 284.91 seconds in (5.5b), and 283.03 seconds in (5.5a). This observation complements our previous conclusion regarding application length and the importance of network programmability for execution acceleration.

While the dynamism of path provisioning plays an essential role for application speed up, shorter timeout values lead to a high number of interruptions, which impact overheads (namely communication with the external controller, as well as path recalculation and provision). Table 5.2 shows the correlation between application runtime, controller interventions, and communication/processing overheads for the LU application. As an example, consider the case in which the timeout value is set up to 10 seconds. The time spent with the mentioned overheads was 6.54 seconds, which represents 2.26% of the total application execution time (283.03 seconds). Even with such overheads, the LU application executed 5.83% faster than the case in which the flow rules are programmed

Figure 5.5 – Application performance as a function of varying rule timeout values for the Sparse – 75% concurrence scenario.



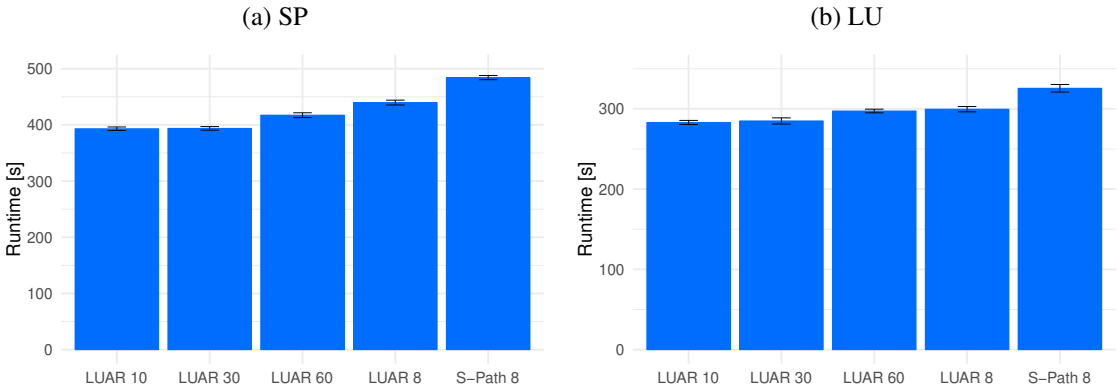
only once (∞ – fourth row of the table).

Table 5.2 – Communication and processing overheads for the LU application incurred by the interruptions to the the external SDN controller.

Timeout (s)	Runtime (s)	Interruptions	Overheads (s)	% Overheads
10	283.03	218	6.54	2.26 %
30	284.91	76	2.50	0.88 %
60	297.30	38	1.10	0.37 %
∞	299.54	10	0.31	0.10 %

Reducing the timeout values indefinitely is not the ideal strategy to take, as the gains in application speed up are increasingly neutralized by the associated overheads (as shown in Table 5.2). Figure 5.6 compares the runtime observed for SP and LU considering distinct idle timeout values. The Shortest Path is used as a baseline. The results obtained in Figures 5.6a and 5.6b confirm that our approach contributes to decreasing application runtime as we reduce the idle timeout values. However, they also suggest that the cost-benefit is higher using LUAR with 30 seconds for the idle timeout. In both situations, 5.6a and 5.6b, when comparing LUAR with idle timeouts set to 30 and 10 seconds, we observe a marginal reduction of application runtime of 0.63 (SP) and 1.88 seconds (LU), at the price of, on average, 2.82X additional reprogramming actions.

Figure 5.6 – Optimal operation point for application speed up observed for the Sparse – 75% concurrence scenario.



6 CONCLUSION

In this thesis, we proposed Janus, a framework for dynamic path provisioning in cloud network infrastructures. It keeps track of link utilization and deploys shortest, least utilized paths to "interconnect" the processes of a running HPC application. The work unfolds into four main research contributions: (*i*) an SDN-centric path provisioning architecture, (*ii*) a link usage-aware routing strategy, (*iii*) a fully functional proof-of-concept implementation, and (*iv*) an in-depth experimental evaluation attesting the framework's efficacy and efficiency.

The results show that the proposed solution has the potential to speed up HPC Applications. LUAR performs particularly well under challenging yet ordinary circumstances such as when VMs are dispersed across the infrastructure, and the network exhibits some degree of contention. Large applications with a high degree of communication between its processes – the norm in high-performance computing – have the potential to benefit the most. For example, LU and SP experiment an average of 13.51% runtime speedup (considering all analyzed factors). Under network contention (*i.e.*, 75%), the gains are, on average, 15.68%, and the best results are as high as 27.01%. To achieve these results, our approach needs to be executed for the first packet of a new flow between two application processes, demanding negligible communication delays with the external SDN controller (less than a dozen milliseconds).

Another critical aspect is properly tunning the idle timeout values for rule expiration. As a general observation, LUAR achieves a higher speed up as idle timeout decreases. However, reducing the timeout values indefinitely must be avoided as there is an inflection point in which path reprogramming times may surpass application speed up margins. For example, in our experiments, the best cost-benefit is achieved using LUAR with 30 seconds for the idle timeout. Under this parameterization, SP and LU reduce runtime by a factor of 0.4%, at the price of 2.82X additional reprogramming actions (compared to the scenario with 10 seconds for the idle timeout).

As future work, we intend to evaluate the effectiveness of the proposed approach to accelerate heavier, real HPC applications being weather forecasting a potential candidate. We also plan improvements to the proposed mechanism that may lead to even higher application runtime speedups. Examples include pre-provisioning paths leveraging potentially known (or expected) application communication patterns and determining the optimal period for flow rules to remain active.

APÊNDICE A – RESUMO EXPANDIDO EM PORTUGUÊS

Aplicações modernas nas áreas da ciência, engenharia e indústria precisam frequentemente lidar com uma quantidade enorme de dados. Essas aplicações precisam resolver desafios relacionados a cenários com grandes volumes de dados, ao mesmo tempo que entregam rapidamente os resultados esperados. Sendo assim, essas aplicações exigem um alto poder computacional e, para isso, a Computação de Alto Desempenho (HPC) torna-se um fator chave para acelerar o processamento de dados. Com o objetivo de atender a esses requisitos computacionais, aplicações de HPC têm tradicionalmente empregado infraestruturas de *clusters*, *grids* e *data centers* (GUPTA et al., 2016). No entanto, recentemente testemunhamos esforços consistentes para a execução de aplicações HPC na nuvem. Essas iniciativas tentam aproveitar características da computação em nuvem como dimensionamento flexível de recursos (elasticidade), disponibilidade instantânea e (comparativo) baixo custo.

Embora a alocação de máquinas virtuais na nuvem (para execução de aplicações HPC) tenha sido amplamente estudada, as taxas de comunicação entre processos ainda representam o principal gargalo de desempenho (LI; ZHANG; LUO, 2017). Além da possibilidade de as máquinas virtuais serem instanciadas fisicamente distantes umas das outras na nuvem, os caminhos provisionados são normalmente estáticos e mais curtos (potencialmente atravessando *links* congestionados e de alto atraso). A natureza dinâmica dos padrões de comunicação observados na maioria das aplicações HPC torna essas limitações ainda mais proeminentes (relevantes). Resumindo, submeter fluxos de HPC a altas latências de rede ainda é um dos principais desafios em aberto, que os ambientes de nuvem precisam enfrentar para oferecer uma infraestrutura adequada para HPC.

Estudos anteriores (EVANGELINOS; HILL, 2008; GUPTA et al., 2016; NETTO et al., 2018b; ROLOFF et al., 2017; WALKER, 2008; WITTE et al., 2020; KOTAS; NAUGHTON; IMAM, 2018) avaliaram a viabilidade do uso de nuvens públicas para HPC. Suas descobertas sugerem que as nuvens não foram projetadas para executar aplicações HPC fortemente acopladas. A principal limitação é o baixo desempenho da rede, resultante da sobrecarga de virtualização de *I/O*, compartilhamento de processador e uso de tecnologias de interconexão de propósito geral. Lee et al. (LEE et al., 2016) e Faizian *et al.* (FAIZIAN et al., 2017) identificaram limitações das tecnologias de rede utilizadas por infraestruturas em nuvem, chamadas de esquemas de encaminhamento simplistas, o que pode resultar na degradação do desempenho de comunicação. Para superar essa limi-

tação, eles propuseram mecanismos de roteamento que evitam caminhos congestionados. No entanto, tais caminhos são calculados antes da execução de uma aplicação HPC e, portanto, não lidam com a flutuação do tráfego de rede ou com padrões de comunicação variáveis.

Nesta dissertação, propomos Janus, um *framework* para provisionamento dinâmico de caminhos *just-in-time* em infraestruturas em nuvem, com o objetivo de acelerar aplicações HPC. Ele sistematicamente (re)programa caminhos para evitar *links* congestionamentos e reduzir o tempo de execução, dados os padrões de comunicação atuais de uma aplicação. Para esse fim, desenvolvemos uma abordagem que monitora continuamente as condições dos *links* de rede para encontrar o caminho menos utilizado entre os nós. Além do *framework*, outra contribuição igualmente importante é a formalização da estratégia *Link Usage-Aware Routing* (LUAR), que desempenha um papel crucial na redução de atrasos de comunicação fim a fim. Também contribuímos fornecendo: (*i*) uma discussão detalhada sobre o estado da arte da literatura acerca do uso de SDNs como um facilitador de infraestruturas em nuvem para executar aplicações HPC; (*ii*) uma descrição de nossa implementação de prova de conceito prototípica; e (*iii*) uma avaliação experimental detalhada da eficácia e eficiência do LUAR. Além de confirmar os benefícios do LUAR sobre o provisionamento de caminho tradicional, discutimos extensivamente os diferentes aspectos que influenciam os ganhos de maior velocidade (fornecendo *insights* para usuários e pesquisadores interessados em melhorar a abordagem proposta).

Os resultados mostram que a solução proposta tem potencial para agilizar a execução das aplicações HPC. O LUAR tem um desempenho particularmente bom em circunstâncias desafiadoras, mas comuns, como quando as VMs estão dispersas pela infraestrutura e a rede exibe algum grau de contenção. Aplicações grandes com alto grau de comunicação entre seus processos – a norma na computação de alto desempenho – têm o potencial de se beneficiar ao máximo. Por exemplo, LU e SP experimentam uma média de 13,51% de aceleração do tempo de execução (considerando todos os fatores analisados). Sob contenção de rede (*i.e.*, 75%), os ganhos são, em média, de 15,68%, e os melhores resultados chegam a 27,01%. Para alcançar esses resultados, nossa abordagem precisa ser executada para o primeiro pacote de um novo fluxo entre dois processos da aplicação, adicionando atrasos de comunicação negligenciáveis com o controlador SDN externo (menos de uma dúzia de milissegundos).

Outro aspecto importante é o ajuste adequado do tempo limite de inatividade para expiração de regras *OpenFlow*. Como observação geral, LUAR atinge uma velocidade

mais alta à medida que o tempo limite de inatividade diminui. No entanto, deve-se evitar reduzir os valores de *timeout* indefinidamente, pois há um ponto de inflexão a partir do qual os tempos de reprogramação dos caminhos podem ultrapassar as margens de aceleração da aplicação. Por exemplo, em nossos experimentos, o melhor custo-benefício é obtido usando LUAR com 30 segundos para o tempo limite de inatividade. Sob esta parametrização, SP e LU reduzem o tempo de execução por um fator de 0,4%, mas com uma redução de 65% nas ações de reprogramação (em comparação com o cenário com 10 segundos para o tempo limite de inatividade).

No futuro, pretendemos avaliar a eficácia da abordagem proposta para acelerar aplicações HPC reais, sendo aplicações na área de previsão do tempo candidatas potenciais. Também planejamos melhorias no mecanismo proposto, que podem levar a acelerações ainda maiores de tempos de execução de aplicações. Exemplos incluem a definição de caminhos pré-provisionados que potencializem padrões de comunicação de aplicações conhecidos (ou esperados) e a determinação do período ideal para que as regras de fluxo permaneçam ativas.

REFERENCES

ABU-LIBDEH, H. et al. Symbiotic routing in future data centers. In: **Proceedings of the ACM SIGCOMM 2010 conference**. [S.l.: s.n.], 2010. p. 51–62.

AL-FARES, M.; LOUKISSAS, A.; VAHDAT, A. A scalable, commodity data center network architecture. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 38, n. 4, p. 63–74, 2008.

AL-FARES, M. et al. Hedera: dynamic flow scheduling for data center networks. In: **Nsdi**. [S.l.: s.n.], 2010. v. 10, n. 8, p. 89–92.

AL-SHABIBI, A. et al. Openvirtex: A network hypervisor. In: **Open Networking Summit 2014 ({ONS} 2014)**. [S.l.: s.n.], 2014.

Alsmadi, I.; Khamaiseh, S.; Xu, D. Network Parallelization in HPC Clusters. In: **International Conference on Computational Science and Computational Intelligence (CSCI'2016)**. [S.l.: s.n.], 2016. p. 584–589.

BAILEY, D. H. NAS Parallel Benchmarks. **Encyclopedia of Parallel Computing**, Springer, p. 1254–1259, 2011.

BALLANI, H. et al. Towards predictable datacenter networks. In: **Proceedings of the ACM SIGCOMM 2011 conference**. [S.l.: s.n.], 2011. p. 242–253.

BENSON, T.; AKELLA, A.; MALTZ, D. A. Network traffic characteristics of data centers in the wild. In: **Proceedings of the 10th ACM SIGCOMM conference on Internet measurement**. [S.l.: s.n.], 2010. p. 267–280.

BERA, S.; MISRA, S.; OBAIDAT, M. S. Mobi-flow: Mobility-aware adaptive flow-rule placement in software-defined access network. **IEEE Transactions on Mobile Computing**, v. 18, n. 8, p. 1831–1842, 2019.

CALVERT, K. Reflections on network architecture: an active networking perspective. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 36, n. 2, p. 27–30, 2006.

CHU, J.; KASHYAP, V. **Transmission of ip over infiniband (ipoib)**. [S.l.], 2006.

CLOS, C. A study of non-blocking switching networks. **Bell System Technical Journal**, Wiley Online Library, v. 32, n. 2, p. 406–424, 1953.

CURTIS, A. R. et al. Rewire: An optimization-based framework for unstructured data center network design. In: IEEE. **2012 Proceedings IEEE INFOCOM**. [S.l.], 2012. p. 1116–1124.

CURTIS, A. R.; KESHAV, S.; LOPEZ-ORTIZ, A. Legup: Using heterogeneity to reduce the cost of data center network upgrades. In: **Proceedings of the 6th International COnference**. [S.l.: s.n.], 2010. p. 1–12.

EVANGELINOS, C.; HILL, C. N. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In: **The 1st Workshop on Cloud Computing and its Applications (CCA)**. [S.l.: s.n.], 2008.

FAIZIAN, P. et al. A comparative study of SDN and adaptive routing on dragonfly networks. In: ACM. **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2017. p. 51.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: an intellectual history of programmable networks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 2, p. 87–98, 2014.

GREENBERG, A. et al. Vl2: a scalable and flexible data center network. In: **Proceedings of the ACM SIGCOMM 2009 conference on Data communication**. [S.l.: s.n.], 2009. p. 51–62.

GUAN, Y. et al. Scalable orchestration of software defined service overlay network for multipath transmission. **Computer Networks**, v. 137, p. 132 – 146, 2018. ISSN 1389-1286. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1389128618301130>.

GUILLEN, L. et al. Sdn-based hybrid server and link load balancing in multipath distributed storage systems. In: **NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2018. p. 1–6.

GUO, C. et al. Bcube: a high performance, server-centric network architecture for modular data centers. In: **Proceedings of the ACM SIGCOMM 2009 conference on Data communication**. [S.l.: s.n.], 2009. p. 63–74.

GUO, C. et al. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In: **Proceedings of the 6th International COnference**. [S.l.: s.n.], 2010. p. 1–12.

GUO, C. et al. Dcell: a scalable and fault-tolerant network structure for data centers. In: **Proceedings of the ACM SIGCOMM 2008 conference on Data communication**. [S.l.: s.n.], 2008. p. 75–86.

GUO, Z.; DUAN, J.; YANG, Y. On-line multicast scheduling with bounded congestion in fat-tree data center networks. **IEEE Journal on Selected Areas in Communications**, IEEE, v. 32, n. 1, p. 102–115, 2013.

GUO, Z. et al. STAR: Preventing flow-table overflow in software-defined networks. **Computer Networks**, Elsevier, v. 125, p. 15–25, 2017.

GUPTA, A. et al. Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud. **IEEE Transactions on Cloud Computing**, v. 4, n. 3, p. 307–321, July 2016. ISSN 2168-7161.

HAUSER, C. B.; PALANIVEL, S. R. Dynamic network scheduler for cloud data centres with sdn. In: **Proceedings of The10th International Conference on Utility and Cloud Computing**. New York, NY, USA: Association for Computing Machinery, 2017. (UCC '17), p. 29–38. ISBN 9781450351492. Available from Internet: <https://doi.org/10.1145/3147213.3147217>.

HOPPS, C. et al. **Analysis of an equal-cost multi-path algorithm**. [S.l.], 2000.

IEEE Standard for Information Technology - Telecommunications and information exchange between systems - Local and Metropolitan Area Networks - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Physical Layer Parameters and Specifications for 1000 Mb/s Operation over 4 pair of Category 5 Balanced Copper Cabling, Type 1000BASE-T. **IEEE Std 802.3ab-1999**, p. 1–144, 1999.

JIA, W.-K. A scalable multicast source routing architecture for data center networks. **IEEE Journal on Selected Areas in Communications**, IEEE, v. 32, n. 1, p. 116–123, 2013.

Joy, S.; Nayak, A. Improving flow completion time for short flows in datacenter networks. In: **IFIP/IEEE International Symposium on Integrated Network Management (IM'2015)**. [S.l.: s.n.], 2015. p. 700–705. ISSN 1573-0077.

KASHYAP, V. Ip over infiniband: Connected mode. **RFC 4755, Internet Engineering Task Force**, 2006.

KOTAS, C.; NAUGHTON, T.; IMAM, N. A comparison of amazon web services and microsoft azure cloud platforms for high performance computing. In: **2018 IEEE International Conference on Consumer Electronics (ICCE)**. [S.l.: s.n.], 2018. p. 1–4.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: ACM. **Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks**. [S.l.], 2010. p. 19.

LEE, J. et al. Enhancing Infiniband with Openflow-style SDN Capability. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. Piscataway, NJ, USA: IEEE Press, 2016. (SC'16), p. 36:1–36:12. ISBN 978-1-4673-8815-3. Available from Internet: <http://dl.acm.org/citation.cfm?id=3014904.3014953>.

LI, C.; ZHANG, J.; LUO, Y. Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm. **Journal of Network and Computer Applications**, Elsevier, v. 87, p. 100–115, 2017.

LI, D. et al. Reliable multicast in data center networks. **IEEE Transactions on Computers**, IEEE, v. 63, n. 8, p. 2011–2024, 2013.

LIAO, J. Sdn system performance. **Online: http://pica8. org/blogs**, 2012.

MAUCH, V.; KUNZE, M.; HILLENBRAND, M. High performance cloud computing. **Future Generation Computer Systems**, Elsevier, v. 29, n. 6, p. 1408–1416, 2013.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 38, n. 2, p. 69–74, 2008.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National . . . , 2011.

MILOJIČIĆ, D.; LLORENTE, I. M.; MONTERO, R. S. Opennebula: A cloud management tool. **IEEE Internet Computing**, IEEE, v. 15, n. 2, p. 11–14, 2011.

MOURA, J.; HUTCHISON, D. Review and analysis of networking challenges in cloud computing. **Journal of Network and Computer Applications**, v. 60, p. 113 – 129, 2016. ISSN 1084-8045. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S108480451500288X>.

NASCIMENTO, M. R. et al. Virtual routers as a service: the routeflow approach leveraging software-defined networks. In: **Proceedings of the 6th International Conference on Future Internet Technologies**. [S.l.: s.n.], 2011. p. 34–37.

NETTO, M. A. et al. Hpc cloud for scientific and business applications: taxonomy, vision, and research challenges. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 1, p. 1–29, 2018.

NETTO, M. A. S. et al. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 51, n. 1, p. 8:1–8:29, jan. 2018. ISSN 0360-0300.

PFISTER, G. F. An introduction to the infiniband architecture. **High performance mass storage and parallel I/O**, v. 42, n. 617-632, p. 102, 2001.

POPA, L. et al. A cost comparison of datacenter network architectures. In: **Proceedings of the 6th International COnference**. [S.l.: s.n.], 2010. p. 1–12.

RADHAKRISHNAN, S. et al. Dahu: Commodity switches for direct connect data center networks. In: IEEE. **Architectures for Networking and Communications Systems**. [S.l.], 2013. p. 59–70.

RAMAKRISHNAN, L. et al. Evaluating interconnect and virtualization performance for high performance computing. **ACM SIGMETRICS Performance Evaluation Review**, ACM, v. 40, n. 2, p. 55–60, 2012.

RATNASAMY, S. et al. A scalable content-addressable network. In: **Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications**. [S.l.: s.n.], 2001. p. 161–172.

ROLOFF, E. et al. Leveraging cloud heterogeneity for cost-efficient execution of parallel applications. In: RIVERA, F. F.; PENA, T. F.; CABALEIRO, J. C. (Ed.). **Euro-Par 2017: Parallel Processing**. Cham: Springer International Publishing, 2017. p. 399–411. ISBN 978-3-319-64203-1.

RYU. Ryu, a component-based software defined networking framework. , 2019. Available from Internet: <RetrievedJanuary26,2019,fromhttp://osrg.github.io/ryu/>.

SEFRAOUI, O.; AISSAOUI, M.; ELEULDJ, M. Openstack: toward an open-source solution for cloud computing. **International Journal of Computer Applications**, Published by Foundation of Computer Science, v. 55, n. 3, p. 38–42, 2012.

SHERWOOD, R. et al. Flowvisor: A network virtualization layer. **OpenFlow Switch Consortium, Tech. Rep**, v. 1, p. 132, 2009.

SPITZNAGEL, E.; TAYLOR, D.; TURNER, J. Packet classification using extended tcams. In: IEEE. **11th IEEE International Conference on Network Protocols, 2003. Proceedings.** [S.l.], 2003. p. 120–131.

SUPERCOMPUTER, T. **TOP500 November 2020 List**. 2020. Available from Internet: <https://www.top500.org/statistics/list/>.

TOKMAKOV, K. et al. A case for data centre traffic management on software programmable ethernet switches. In: **2019 IEEE 8th International Conference on Cloud Networking (CloudNet)**. [S.l.: s.n.], 2019. p. 1–6.

WALKER, E. Benchmarking Amazon EC2 for Hig-Performance Scientific Computing. **;login:: The magazine of USENIX & SAGE**, USENIX Association, v. 33, n. 5, p. 18–23, 2008.

WITTE, P. A. et al. An event-driven approach to serverless seismic imaging in the cloud. **IEEE Transactions on Parallel and Distributed Systems**, v. 31, n. 9, p. 2032–2049, 2020.

YANG, C. et al. Big data and cloud computing: innovation opportunities and challenges. **International Journal of Digital Earth**, Taylor & Francis, v. 10, n. 1, p. 13–53, 2017. Available from Internet: <https://doi.org/10.1080/17538947.2016.1239771>.

Zahid, F. et al. A self-adaptive network for hpc clouds: Architecture, framework, and implementation. **IEEE Transactions on Parallel and Distributed Systems**, v. 29, n. 12, p. 2658–2671, 2018.