Federal University of Parana

Celio Trois

Communication Patterns Abstractions for Programming SDN to Optimize High-Performance Computing Applications

Curitiba PR
2017

Celio Trois

Communication Patterns Abstractions for Programming SDN to Optimize High-Performance Computing Applications

Final Doctoral Thesis presented as partial requirement to obtain a Ph.D. in Computer Science at the Federal University of Parana (UFPR).

Area: *Computer Science*.

Supervisor: Luis Carlos Erpen de Bona.

Co-supervisors: Magnos Martinello.
Marcos Didonet Del Fabro.

Curitiba PR
2017

T845c Trois, Celio
Communication Patterns Abstractions for Programming SDN to Optimize High-Performance Computing Applications / Celio Trois. – Curitiba, 2017.
114 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2017.

Orientador: Luis Carlos Erpen de Bona – Co-orientador: Magnos Martinello – Co-orientador: Marcos Didonet Del Fabro.

Bibliografia: p. 97-114.

1. Redes de computadores. 2. Sistemas de computação em rede. 3. Computação de alto desempenho. I. Universidade Federal do Paraná. II.Bona, Luis Carlos Erpen de. III. Martinello, Magnos. IV. Del Fabro, Marcos Didonet. V. Título.

CDD: 004.66

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós Graduação em INFORMÁTICA
Código CAPES: 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Tese de Doutorado de **CELIO TROIS**, intitulada: **"Communication Patterns Abstractions for Programming SDN to Optimize High-Performance Computing Applications"**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 04 de Setembro de 2017.

LUIS CARLOS ERPEN DE BONA
Presidente da Banca Examinadora (UFPR)

DANIEL WEINGAERTNER
Avaliador Interno (UFPR)

EDUARDO PARENTE RIBEIRO
Avaliador Externo (UFPR)

LUIZ EDUARDO SOARES DE OLIVEIRA
Avaliador Interno (UFPR)

CESAR AUGUSTO CAVALHEIRO MARCONDES
Avaliador Externo (UFSCAR)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

to Luciana, Davi, and Artur.

Acknoledgements

Firstly, I would like to thank my family, especially my wife, Luciana, who has always helped me and encouraged me; my son Davi, who accompanied my doctoral work from the first to the fifth year of his life; and my son Artur, who was born in the middle of my doctorate. I remember fondly the many times my children sat next to me and while I worked, they drew, asking me to incorporate their drawings into the articles I was writing. I lost count of the times I had to stop my work to patiently explain to them the meaning of the little squares I was stared at. My mother Cleonice, my brother Sergio, my sister Sonia, my mother-in-law Terezinha, my father-in-law Dirceu, thank you!

I am grateful to my advisor, Bona, for the advice and valuable tips (and the IPAs) during the four years that I have developed this work. I also appreciate my co-advisors for the meetings and debates; Didonet, a great friend since our graduation and Magnos who went from a stranger, at the time of writing the proposal, to a close friend at the end of this work. It is impossible to name all those people who, in one way or another, contributed to the completion of this thesis, but I extend to all my grateful thanks.

Finally, I would like to thank CAPES for partial funding of this research, CNPq under Grant no. 456143/2014-9, and the Brazilian Ministry of Communications for partial funding it via "Digital Inclusion: Technology for Digital Cities" project. I also thank the European Commission H2020 program under grant agreement no. 688941 (FUTEBOL), as well from the Brazilian Ministry of Science, Technology, Innovation, and Communication (MCTIC) through RNP, CTIC and FAPES (under grants no. 0410/2015 and no. 0444/2015).

Resumo

A evolução da computação e das redes permitiu que múltiplos computadores fossem interconectados, agregando seus poderes de processamento para formar uma computação de alto desempenho (HPC). As aplicações que são executadas nesses ambientes processam enormes quantidades de informação, podendo levar várias horas ou até dias para completar suas execuções, motivando pesquisadores de varias áreas computacionais a estudar diferentes maneiras para acelerá-las. Durante o processamento, essas aplicações trocam grandes quantidades de dados entre os computadores, fazendo que a rede se torne um gargalo. A rede era considerada um recurso estático, não permitindo modificações dinâmicas para otimizar seus links ou dispositivos. Porém, as redes definidas por software (SDN) emergiram como um novo paradigma, permitindo-a ser reprogramada de acordo com os requisitos dos usuários. SDN já foi usado para otimizar a rede para aplicações HPC específicas mas nenhum trabalho tira proveito dos padrões de comunicação expressos por elas. Então, o principal objetivo desta tese é pesquisar como esses padrões podem ser usados para ajustar a rede, criando novas abstrações para programá-la, visando acelerar as aplicações HPC. Para atingir esse objetivo, nós primeiramente pesquisamos todos os níveis de programabilidade do SDN. Este estudo resultou na nossa primeira contribuição, a criação de uma taxonomia para agrupar as abstrações de alto nível oferecidas pelas linguagens de programação SDN. Em seguida, nós investigamos os padrões de comunicação das aplicações HPC, observando seus comportamentos espaciais e temporais através da análise de suas matrizes de tráfego (TMs). Concluímos que as TMs podem representar as comunicações, além disso, percebemos que as aplicações tendem a transmitir as mesmas quantidades de dados entre os mesmos nós computacionais. A segunda contribuição desta tese é o desenvolvimento de um framework que permite evitar os fatores da rede que podem degradar o desempenho das aplicações, tais como, sobrecarga imposta pela topologia, o desbalanceamento na utilização dos links e problemas introduzidos pela programabilidade do SDN. O framework disponibiliza uma API e mantém uma base de dados de TMs, uma para cada padrão de comunicação, anotadas com restrições de largura de banda e latência. Essas informações são usadas para reprogramar os dispositivos da rede, alocando uniformemente as comunicações nos caminhos da rede. Essa abordagem reduziu o tempo de execução de benchmarks e aplicações reais em até 26.5%. Para evitar que o código da aplicação fosse modificado, como terceira contribuição, desenvolvemos um método para identificar automaticamente os padrões de comunicação. Esse método gera texturas visuais differentes para cada TM e, através de técnicas de aprendizagem de máquina (ML), identifica as aplicações que estão usando a rede. Em nossos experimentos, o método conseguiu uma taxa de acerto superior a 98%. Finalmente, nós incorporamos esse método ao framework, criando uma abstração que permite programar a rede sem a necessidade de alterar as aplicações HPC, diminuindo em média 15.8% seus tempos de execução.

Palavras-chave: Redes Definidas por Software, Padrões de Comunicação, Aplicações HPC.

Abstract

The evolution of computing and networking allowed multiple computers to be interconnected, aggregating their processing powers to form a high-performance computing (HPC). Applications that run in these computational environments process huge amounts of information, taking several hours or even days to complete their executions, motivating researchers from various computational fields to study different ways for accelerating them. During the processing, these applications exchange large amounts of data among the computers, causing the network to become a bottleneck. The network was considered a static resource, not allowing dynamic adjustments for optimizing its links or devices. However, Software-Defined Networking (SDN) emerged as a new paradigm, allowing the network to be reprogrammed according to users' requirements. SDN has already been used to optimize the network for specific HPC applications, but no existing work takes advantage of the communication patterns expressed by those applications. So, the main objective of this thesis is to research how these patterns can be used for tuning the network, creating new abstractions for programming it, aiming to speed up HPC applications. To achieve this goal, we first surveyed all SDN programmability levels. This study resulted in our first contribution, the creation of a taxonomy for grouping the high-level abstractions offered by SDN programming languages. Next, we investigated the communication patterns of HPC applications, observing their spatial and temporal behaviors by analyzing their traffic matrices (TMs). We conclude that TMs can represent the communications, furthermore, we realize that the applications tend to transmit the same amount of data among the same computational nodes. The second contribution of this thesis is the development of a framework for avoiding the network factors that can degrade the performance of applications, such as topology overhead, unbalanced links, and issues introduced by the SDN programmability. The framework provides an API and maintains a database of TMs, one for each communication pattern, annotated with bandwidth and latency constraints. This information is used to reprogram network devices, evenly placing the communications on the network paths. This approach reduced the execution time of benchmarks and real applications up to 26.5%. To prevent the application's source code to be modified, as a third contribution of our work, we developed a method to automatically identify the communication patterns. This method generates different visual textures for each TM and, through machine learning (ML) techniques, identifies the applications using the network. In our experiments the method succeeded with an accuracy rate over 98%. Finally, we incorporate this method into the framework, creating an abstraction that allows programming the network without changing the HPC applications, reducing on average 15.8% their execution times.

Keywords: Software-Defined Networking, Communication Patterns, HPC Applications.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Program Interface

APS average packet size

CDPI Control-Data-Plane Interface

CPNP Communication Pattern Network Programming

DPI Deep Packet Inspection

DSL Domain-Specific Language

FODA Feature-Oriented Domain Analysis

FPGA field-programmable gate array

FRP Functional Reactive Programming

FTP File Transfer Protocol

GPL General-purpose Language

GPU graphics processing unit

HPC High-Performance Computing

IDS Intrusion Detection System

JSON JavaScript Object Notation

LACP Link Aggregation Control Protocol

LBP Local Binary Pattern

LLDP Link Layer Discovery Protocol

ML Machine Learning

MPI Message Passing Interface

NBI Northbound Interface

NPB NAS Parallel Benchmarks

NPM Network Programming Module

OMB OSU Micro-Benchmarks

ONF Open Networking Foundation

OVSDB Open vSwitch Database

PPM Path Processor Module

QoS Quality of Service

TM traffic matrix

ToR top-of-rack

ULBP Uniform LBP

REST Representational State Transfer

RF Random Forest

RGM Rules Generator Module

RLBP Robust LBP

RTT round-trip time

SDN Software-Defined Networking

SDK Software Development Kit

SLA Service Layer Abstraction

SNMP Simple Network Management Protocol

SVM Support Vector Machines

TCAM Ternary Content Addressable Memories

TIM Topology Information Module

TMD Traffic Matrix Database

VM virtual machine

XML Extensible Markup Language

Chapter 1

Introduction

In the last few decades, the computing power evolved from mono-processor computers to many thousands of interconnected multiprocessor computers. This new scenario of aggregating multiple computing nodes for working as a supercomputer is called High-Performance Computing (HPC). HPC applications are being used in academia and laboratories for scientific research and in industries for business and analytics [Gupta and Milojicic, 2011]. Scientists are using these applications to create and predict complex phenoms, for example, weather forecasting, prediction of natural disasters, bacterial profiling, animal genotyping, and so forth. HPC has also a wide range of business applications, such as, analyzing large volumes of customer data or logs from monitoring real-time network, simulating product designs, modeling complex workflows, and exploring many aspects of social networks. For improving the quality of their results, HPC applications are increasingly demanding computational power, being executed in dedicated clusters, moving huge volumes of data, and taking hours, or even days, to complete their executions.

Due to the considerable time that HPC applications take for finishing their executions, many researchers are studying and presenting proposals for accelerating them. The application's performance may be affected by several aspects, creating multiple study opportunities and making this area widely researched. There are works modifying scheduler for optimizing the job placement [Renner et al., 2015, Al-Fares et al., 2010, Lee et al., 2014], using field-programmable gate arrays (FPGAs) [Dimond et al., 2011, Vassiliev, 2017] or graphics processing units (GPUs) [Erlacher et al., , Tangherloni et al., 2017], proposing improved programming languages and frameworks [Nugteren, 2017, Kuster, 2017], employing cache [Yu et al., 2017, Gémieux et al., 2017], taking advantage of cloud resources [Righi et al., 2016, Galante et al., 2016], and so forth.

Another aspect used for improving the HPC applications is the fact that the wide majority is implemented using well-known numerical methods, so-called dwarfs [Asanovic et al., 2006]. A computational dwarf can be defined as "a pattern of communication and computation common across a set of applications". These communication and computation patterns are also intensely researched and have been used for optimizing the communication on networks-on-chip [Werner et al., 2017], GPUs [Wang et al., 2016], multiprocessor architectures [Prabhakar et al., 2017], and ameliorating the performance of applications [Rubin et al., 2014].

The network is also widely studied for improving the performance of these applications. The main reason is that the network performance did not evolve at the same rate that other HPC resources, becoming a bottleneck. For example, comparing the network transmission rate with the individual node computational power, in last decade, the com-

putational power grew 36 times more than the network capacity [Jain, 2016]. So, understanding the communication demands of HPC applications and classifying them is a challenge in networking research [Srivastava et al., 2016] because it can be instrumental in several management activities, such as monitoring [Chowdhury et al., 2014, Van Adrichem et al., 2014, Yu et al., 2014], expansion planning [Simmons, 2014], traffic engineering [Trestian et al., 2013, Wang et al., 2008, Akyildiz et al., 2014], detection of anomalies [Giotis et al., 2014], and energy saving [Yao et al., 2016].

Optimizing the communication of HPC applications is also challenging due to the limited control over the system environment. Unlike the computation resources that are typically known a priori and are fully under the control of an executing application, the availability of network resources may neither be predictable nor be completely under the control of an executing application [Jain, 2016]. The common assumption is that HPC applications run on a fixed number of processes, where the network is considered a static resource, working as a connectivity service that cannot be controlled or modified [Righi et al., 2016]. Moreover, the current network stack provides a best-effort service model for hosts communication, which is not enough to satisfy the on-demand data access required by these applications [Sakr et al., 2011]. For combating these problems, Software-Defined Networking (SDN) emerged as an architecture decoupling the network control and forwarding functions, enabling the network to become directly programmable, and the underlying infrastructure being abstracted for applications and services [ONF, 2013]. One benefit offered by SDN is that the network can be modified according to the user requirements, inclusive for suiting transient demands, allowing runtime adjustments for improving the performance of specific applications.

The SDN architecture is divided into three main layers: Data Plane, Control Plane, and Application Plane [ONF, 2013]. The network devices (Data Plane) are managed by a remote and decoupled Control Plane. The Control Plane observes and controls the whole network from a logically centralized point, offering functions such as access control, orchestration, and network virtualization. The network functions (e.g. routing, load balancer, firewall) are performed by the network applications in the Application Plane. OpenFlow [McKeown et al., 2008] is a de facto standard interface between Data Plane and Control Plane, but there is no standardization on the interface between Control Plane and Application Plane. As this interface is used for creating the applications that will modify the network behavior, the lack of standardization makes it hard for researchers and developers to know which features and abstractions offered by SDN can be useful to them. The interface between Control Plane and Application Plane can also be implemented as high-level SDN programming languages; however, they also do not present any kind of standardization, focusing mainly on network policies and offering abstractions focusing network operators.

HPC is one of the most resource-intensive computing application areas, and therefore SDN can effectively serve HPC [Kawai, 2012]. Different techniques proposed by SDN are employed to optimize the network for specific HPC applications, for example, dynamically applying Quality of Service (QoS) on network paths [Egilmez et al., 2012, Narayan et al., 2012], modifying the forwarding [Wang et al., 2012a, Kumar et al., 2015, Tang et al., 2017], reading information from the network for improving the jobs placement [Qin et al., 2014, Renner et al., 2015, Lee et al., 2014], modifying the topology [Vassoler et al., 2012], and so forth.

Considering that HPC applications present well-known communication patterns and the flexibility offered by SDN for modifying the network dynamically, this thesis proposes the development of a high-level abstraction using the communication patterns for programming the network, aiming to speed up HPC applications. So, the main research question addressed in this

thesis is: *Is it possible to speed up the HPC applications by tuning the network according to their communication patterns?*

To answer this question, we first studied the SDN architecture, analyzing the different levels for programming the SDN-enabled devices. The first contribution of our research is a survey on the SDN programming languages, where all languages' prominent features were mapped and the similar ones were grouped, leading the creation of a taxonomy. This study gave us a thorough understanding of which abstractions offered by SDN could be effective for improving the performance of HPC applications. We also paralleled the advantages and issues introduced by SDN, relating them with works aiming to accelerate HPC applications.

We researched works identifying the HPC applications' communication patterns and their requirements. For a better comprehension on these patterns, we inspected the spatial and temporal behaviors of HPC applications by analyzing their traffic matrixs (TMs), varying the input size and the number of computing nodes. This observation enabled us to conclude that (i) HPC applications present "well-behaved" communications, tending to transmit the same amount of information among the same computational nodes and (ii) the TM can be used to represent the communication patterns of HPC applications. We also revisited the existing techniques for computationally classifying the applications, realizing that the communications can be identified by port-based, Deep Packet Inspection (DPI), and Machine Learning (ML)-based techniques, each presenting advantages and issues.

For understanding the network factors that may slow down the performance of HPC applications, we studied the impact caused by the delay imposed by the network topology and the unbalancing in the load of its links. We also investigated the effects of the intrinsic SDN programmability on the applications' execution time, concluding that the main issues were the latency for populating the switches' flow tables and the time for performing a lookup on them. So, the second contribution of this thesis is the development of a framework designed to prevent the issues introduced by SDN. It decreases the latency for populating the flow tables by proactively installing the rules before starting the application, the flow table lookup time is reduced by grouping the matching flows. The framework keeps a database of communication patterns represented by the TMs and annotated with latency-sensitive and bandwidth-intensive constraints. This database is used for placing the application's communications on the network paths, forwarding the latency-sensitive through low-latency paths and using multiple paths for balancing the bandwidth-intensive. We tested this framework on HPC benchmarks and applications, achieving a reduction up to 26.5% in the execution time of real applications and at best, it halved the benchmark execution time.

The proposed framework is intrusive, requiring the application source code to be modified by including calls to its Application Program Interface (API). This led us to implement an autonomic solution for detecting the HPC applications communication patterns. Due to characteristics of HPC applications communications, such as encryption and dynamically assigned ports, we decided using an ML-based method. So, another contribution made during this thesis consists of a novel methodology using features extracted from TMs as input for the ML classifiers. This method applies a function for rendering different visual textures for each communication pattern; after that, well-known textural representations are used for extracting the feature vectors to feed different classifiers. We performed experiments for testing the classification, comparing the accuracy of our proposal with four methods described in current literature. The results showed that the accuracy of our method is less than 2% than the optimal solution, while the best accuracy obtained with the other methods was 10.5% worse than ours.

Finally, we extended the framework, implementing our ML method on it, creating a higher-level abstraction for programming the network. The framework has two modes of

operation: learning and running. In learning mode it collects TMs from the network, applies the ML method, and stores the computed information to feed the classifier; in this mode, the user must inform the latency and bandwidth constraints. In the running mode, at each predetermined interval of time, the framework acquires the TM from the network, applies the ML method, and classifies the communication pattern. If the same communication pattern is classified by a predetermined number of times, the framework reprograms the network for that pattern. To evaluate the framework, we first set up the network with Link Aggregation Control Protocol (LACP), a well-known protocol for load balancing across multiple links, and then we executed a random queue of benchmarks, measuring its execution time. Next, we fed the framework with the benchmarks' communication patterns and executed it in the running mode for classifying the communications and modifying the network behavior. The measured time for executing the queue with our approach was 15.8% lower than LACP.

The remainder of this thesis is organized as follows, Chapter 2 shows our survey on SDN architecture, reporting its different programming levels, the features and abstractions of SDN programming languages, and our proposed taxonomy for classifying them. Chapter 3 presents the advantages and issues introduced by SDN, relating them with works using SDN for optimizing HPC applications. The communication patterns, the spatial and temporal behaviors investigation, and the existing techniques for automatically classifying them are also reported in Chapter 2. Chapter 4 describes our proposed framework that uses the communication patterns as the key logic for programming the network and its evaluation and results. Chapter 5 relates our ML method and its implementation in the framework, also presenting its assessments and results. Finally, Chapter 6 provides the concluding remarks of this thesis and outlines the potential future work.

Chapter 2

SDN: Overview, Abstractions, and Classification

SDN is a new concept of networks where the forwarding devices are dumb, and the routing algorithms are implemented in a separated entity called controller; the controller is responsible for programming the forwarding decisions of these dumb devices. SDN is flexible, allowing the network behavior to be adjusted on-the-fly for meeting specific requirements, so it can be useful for optimizing the performance of HPC applications. In this chapter, we first introduce the SDN architecture, explaining its components, layers, and interfaces. Next, we report through examples all the possible levels of abstraction for programming these networks. Finally, we present an in-depth study of the existing abstractions at the highest level, the SDN programming languages. We believe that a proper understanding of these abstractions is imperative for achieving the objectives proposed in this thesis.

SDN aims to overcome the conventional network infrastructure limitations where each device has a specific rule on the network. For example, switches are used to connect computers inside a Local Area Network (LAN); routers interconnect several LANs and/or to connect the network to the Internet; firewalls filter out unwanted packets, and so forth. Furthermore, each device must be individually configured, eventually causing misconfigurations and errors, such as forwarding loops, packet losses, unintended paths, and contract violations [Feamster and Balakrishnan, 2005]. A further problem is the development and deployment of new networking functions or protocols. Most of the vendors provide "black box" devices, not allowing the network administrator to modify the embedded software. Certain suppliers offer Software Development Kits (SDKs) that allow certain flexibility but, nevertheless, it is necessary to have all devices from this same vendor, and also carefully install and configure new firmware on all devices individually.

SDN, in contrast, is defined to be a network architecture where the forwarding devices (Data Plane) are managed by a remote and decoupled Control Plane [Lantz et al., 2010]. The Control Plane observes and controls the whole network from a (logically) centralized point, offering functions such as routing protocols, access control, network virtualization, power management, and also allowing the implementation of new protocols. One outcome introduced by SDN is the possibility of a network to be defined/modified after it has been implemented. New features can be added without modifying the hardware, allowing the network to evolve at the same speed as the software [Lantz et al., 2010]. Figure 2.1 shows a comparison between the conventional network *versus* SDN infrastructures. In the top of the figure, it is possible to note different devices such as switches, load balancer, router, and firewall. In the bottom of the figure, it is possible to see the same infrastructure, but with the Data Plane separated from Control Plane

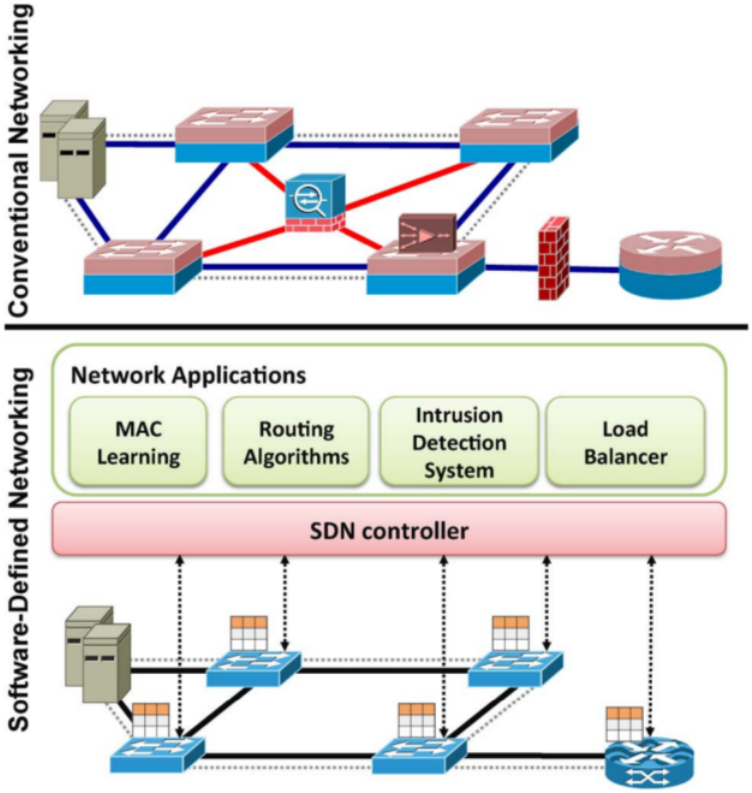


Figure 2.1: Conventional networking *versus* SDN infrastructures. Source: [Kreutz et al., 2015]

(SDN Controller). The network functions (routing, load balancer), in this case, are performed by the network applications. These applications can be easily modified without any modification on the network.

2.1 Architecture

The Open Networking Foundation (ONF) is a non-profit organization dedicated to the development and standardization of Software-Defined Networking. ONF presented a high-level view of SDN architecture [ONF, 2013], dividing it into three main layers: Data Plane, Control Plane, and Application Plane. Figure 2.2[1] shows a graphical representation of the architectural components and their interactions.

2.1.1 Data Plane

Data Plane layer (also known as Forwarding Plane) is comprised of one or more Network Elements. Each Network Element contains one or more Datapaths. The Datapath is a network device, which enables visibility and control over forwarding and data processing, consists of one or more Forwarding Engines. Usually, the Forwarding Engine uses a matching table to lookup the destination address of incoming packets and forwards them to the proper outgoing port(s). In the scope of this thesis, the terms forwarding devices, network devices, and switches are interleaved, referring to the Datapaths.

[1]Source: [ONF, 2013]

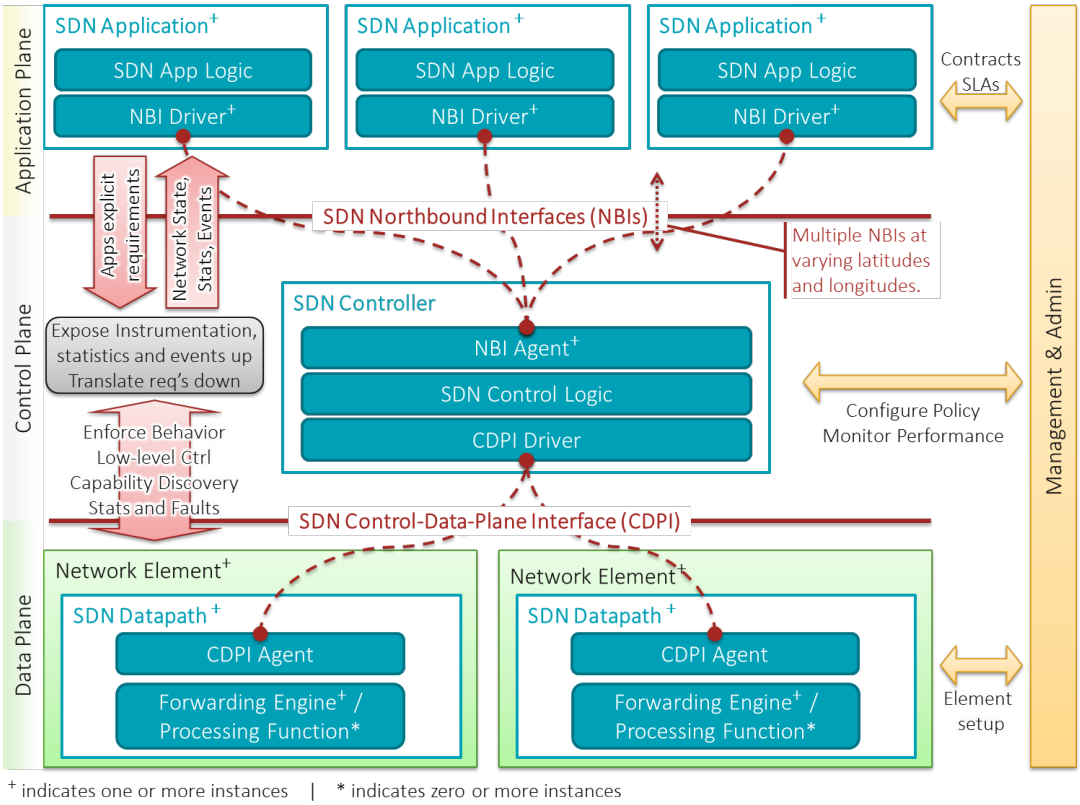


Figure 2.2: SDN architecture.

A Datapath implements a Control-Data-Plane Interface (CDPI) Agent to communicate with Control Plane layer. Sometimes, it also implements Processing Functions to perform internal traffic processing, for example, DPI [Bouet et al., 2013]. The Datapath logical definition does not prescribe implementation details such as the logical to physical mapping, management of shared physical resources, virtualization or slicing, interoperability with non-SDN networking, nor the data processing functionality [ONF, 2013]. Moreover, one Datapath may also be defined across multiple physical network elements [Kang et al., 2013].

Data Plane also has an interface to communicate the Data Plane with the Control Plane. This layer is responsible for forwarding (and/or modifying) all the network traffic. The Data Plane handles all network packets through several operations, where the typical operations are lookup, forward (in particular cases multicast or replicate), drop, re-mark, count, and queue.

Virtualization is also part of Data Plane layer. Many virtual switches can be defined inside a real device [Drutskoy et al., 2013], or one virtual switch can be instantiated across multiple physical devices [Kang et al., 2013]. Virtualization allows multiple logical networks being shared by the same physical network [Jain and Paul, 2013].

Control-Data-Plane Interface

The term Control-Data-Plane Interface (CDPI) is also called as "Southbound API" and refers to a communication interface between Data and Control Planes. This interface specifies operations to manage and control the networking devices. Besides the commands to manage flow tables, and it also provides statistical information on link usage, forwarding tables, ports, queues, and so forth.

To date, OpenFlow [McKeown et al., 2008] is the most used CDPI, but it is not the only one. There exist some other proposals such as ForCES [Doria et al., 2010], POF [Song, 2013], Open vSwitch Database (OVSDB) [Pfaff and Davie, 2013], OpFlex [Smith et al., 2014], and POSEIDON [Hakiri and Gokhale, 2016]. OpenFlow was originally designed and implemented to "*allow researchers to run experimental protocols in the networks they use every day*" [McKeown et al., 2008]. OpenFlow is based on the Ethernet switch, with an internal flow table, and a standardized interface to add and remove flow entries.

The OpenFlow protocol is the first standard communication interface [Ren and Xu, 2014] of SDN era. It is also the most accepted and deployed CDPI [Kreutz et al., 2015]. OpenFlow presents an open interface to allow Data Plane forwarding tables to be managed, by adding, modifying and removing matching rules and actions. The OpenFlow version 1.0 primitives are presented in Figure 2.3[2]. OpenFlow-enabled switches have a flow table with a set of matching fields (used to match against packet header values), statistical counters, and a set of zero or more actions to be applied in the case of matching.

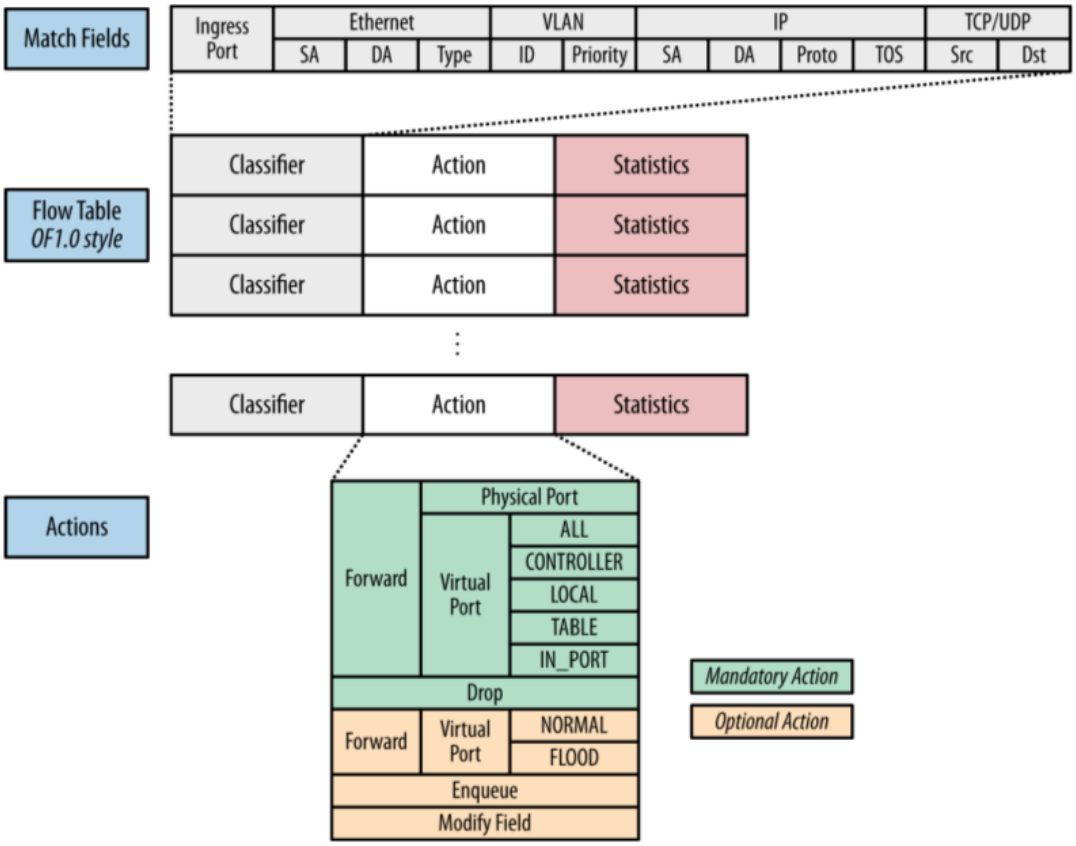


Figure 2.3: OpenFlow v1.0 primitives.

In the basic OpenFlow operation, upon receiving a new packet, the network device compares the packet header fields against its flow table entries. If a matching entry is found, the actions set on that entry are performed on the packet (e.g., the action might be to forward a packet out a specified port). If no match is found, the packet is forwarded to the controller through OpenFlow protocol. The controller manages the switch's flow table by adding and/or removing the flow entries (once again using OpenFlow protocol). The routing decisions can be

[2]Source: [Nadeau and Gray, 2013]

made periodically or "ad hoc" by the controller; they are translated into rules and actions that are installed in the switches' flow tables with a configurable lifespan and, during this time, the forwarding is performed at wire speed [McKeown et al., 2008].

OpenFlow was firstly proposed by Nick McKeown in April 2008. In March 2011 was founded the ONF as a non-profit, user-driven organization dedicated for promoting SDN through open standards. ONF adopted OpenFlow, releasing protocol specification versions numbered from 1.0 to 1.5, introducing many features and improvements; the last version was released in 2014.

Network Elements

The Network Elements are composed of one or more Datapaths. These elements have Forwarding Engine(s), as the name suggest, are used to forward the packets through their ports. Usually, this Engine has a matching table where the incoming packets are compared with. If a match occurs, the actions (forward, drop, modify fields, enqueue) registered in that match line are performed (as can be seen in Figure 2.3). Besides the forwarding version introduced by OpenFlow, there are also other works proposing different Forwarding Engines [Ramos et al., 2013, Vencioneck et al., 2014, Martinello et al., 2014].

Many manufacturers (e.g., Hewlett-Packard, Extreme Networks, Huawei, Juniper) are launching SDN-compatible switches. Most of these devices have implemented in hardware an architecture compatible with OpenFlow v1.0 and, some of them already have a certification issued by ONF[3], for example HP 2920, HP 3800, HP 5420, NEC PF5240. ONF released many versions of OpenFlow, with significant changes, in a short period of time, forcing the manufacturers to modify their strategy on enabling the instantiation of virtual SDN-switches in their physical switches.

Virtualization

The frequent changes in the OpenFlow specification forced several researchers and companies develop software switches. Besides this, Jain [Jain and Paul, 2013] showed five more reasons to virtualize resources:

Sharing: When a resource is too big for a single user, it is best to divide it into multiple virtual pieces, as is the case with today's multi-core servers. Each server can run multiple virtual machines (VMs), and each machine can be used by a different user. The same applies to high-speed links and large-capacity disks. All virtual switch presented in Table 2.1 permits this kind of sharing.

Isolation: Multiple users sharing a resource may not trust each other, so it is important to provide isolation among users. Users using one virtual component should not be able to monitor or interfere the activities of other users. Flowvisor [Sherwood et al., 2009] is the most known technology that permits isolation, but there exist other initiatives focusing on isolation such as AutoVFlow [Yamanaka et al., 2014], OpenVirteX [Al-Shabibi et al., 2014], AutoSlice [Bozakov and Papadimitriou, 2012], and SD-NMS [Ahmed et al., 2015].

Aggregation: If the resources are small, it is possible to construct a large virtual resource by grouping them. It is the case of storage, where a large number of inexpensive unreliable disks can be used to make up large reliable storage. Usually, aggregation is called "Big Switch." Kang et al. [Kang et al., 2013] and Yan et al. [Yan et al., 2017] are examples of abstractions for creating big switches.

[3] https://www.opennetworking.org/openflow-conformance-certified-products

Table 2.1: OpenFlow software switches.

Device	OpenFlow Version	Features
Indigo [Floodlight, 2017]	1.0	runs on physical switches and uses features and takes advantage of the ASICs
LINC [Linc, 2017]	1.2-1.3	multiple logical switches, runs on Linux, Solaris, Windows, MacOS, and FreeBSD
of13softswitch [CPqD, 2017]	1.3	user-space software switch based on Ericsson TrafficLab 1.1 softswitch[4]
Open vSwitch [Linux, 2017]	1.0-1.3, 1.4*, 1.5*	ported to multiple virtualization platforms and switching chipsets; integrated into many virtual management systems; support distribution across multiple physical devices
XORPlus [Xorplus, 2017]	1.0	software switch to high-performance ASICs
* Supported, with one or more missing features.		

Dynamics: Often resource requirements change fast due to, for example, user mobility. It is necessary to provide a way to quickly reallocate the resources. Virtual resources make it easier than with physical resources. As example, we can cite Keller et al. [Keller et al., 2012] and Yao et al. [Yao et al., 2016] proposing solution for live migration and VM placement for optimize the network and reduce energy costs.

Ease of management: Last but not least is the ease of management. Virtual devices are easier to manage because they are software-based and expose a uniform interface through standard abstractions. Gember et al. [Gember et al., 2012] presented one example of simplifying middlebox management through SDN.

Virtual switches use the same idea of VMs for sharing the network resources. Table 2.1 shows examples of open-source software switch implementations, the supported OpenFlow version, and some relevant features.

As mentioned before, the Data Plane uses the CDPI to communicate with the upper layer called Control Plane. Next section explains the Control Plane layer, showing its main element (Controller) and the interface Northbound Interface (NBI) which enables the controller to be programmed with a higher level of abstraction.

2.1.2 Control Plane

Control Plane layer is composed of the Controller and the Northbound Interface. The controller is responsible for maintaining all the network paths, as well as programming each network device using the commands and responses described in the CDPI. The controller exposes and abstracts network functions and operations via NBI programmatic interfaces. These functions can include orchestration [Autenrieth et al., 2013, Banikazemi et al., 2013, Kotronis et al., 2014], mediation [Yin et al., 2014], and also provide interface for user applications to exchange information with the Network Elements.

The functions offered by the Control Plane are provided through Representational State Transfer (REST) [Jakl, 2008] or proprietary APIs. These APIs allow the information to be exchanged directly with the Controller. NBI can also be defined as a network

[4] https://github.com/TrafficLab/of11softswitch

programming language (e.g., Frenetic [Foster et al., 2010], FML [Hinrichs et al., 2009], NetCore [Monsanto et al., 2012]) implemented on the top of Controller, facilitating the development of network applications.

Controllers

One of the main advantages presented by SDN paradigm is having a logically centralized view of the network through Controller. This centralized view eliminates the need for distributed network protocols, which is typically complicated and sometimes do not estimates the best routing solution [Goransson and Black, 2014]. Controllers are also called Network Operating Systems, and they are the main element in Control Plane, carrying out all the network "intelligence."

Controllers support the SDN Applications, translating them to network configuration based on the policies defined by the network operator. They present two behaviors: reactive and proactive. By acting reactively, the first packet of a flow received by a network device is sent to the Controller. The Controller, based on its Application logic, inserts or modifies flow entries on the switches' flow tables. This approach presents the most efficient use of flow tables, but every new flow causes an additional setup time. In the proactive approach, the controller pre-populates the flow tables of all network devices, and no additional flow setup time is needed. However, in this case, it is necessary to aggregate specify rules to cover all routes, and it is harder to deal with runtime network modifications [Jammal et al., 2014].

The first SDN Controller was NOX [Gude et al., 2008]. In 2008, NOX was adopted by the SDN community and became open source. It was used as the basis for some subsequent Controllers, such as Onix [Koponen et al., 2010] and POX[5]. Beacon [Erickson, 2013] is a Java-based OpenFlow Controller created in early 2010; Floodlight [Networks, 2015] was forked from Beacon. There are some other Controllers, such as Trema (Ruby-based) [Takamiya, 2015], and Ryu [Ryu, 2015]. Network vendors, such as Cisco, HP, IBM, VMWare, and Juniper created their own Controllers, most of them based were based on Beacon, but lately, they moved toward OpenDaylight [Medved et al., 2014]. The OpenDaylight Controller is Java-based also derived from the original Beacon, currently supported by the Linux Foundation.

Table 2.2: OpenFlow controllers.

Controller	NBI	CDPI	OF Version	Language
Beacon	proprietary	OpenFlow	1.0	Java
Floodlight	RESTful	OpenFlow	1.0-1.4	Java
HP VAN	RESTful	OpenFlow, L2 & L3 agents	1.0, 1.3	Java
NOX	proprietary	OpenFlow	1.0	C++
Onix	proprietary	OpenFlow	1.0	C, Python
OpenDayLight	REST	SLA	1.0, 1.3	Java
POX	proprietary	OpenFlow	1.0	Python
Ryu	proprietary	OpenFlow	1.0, 1.2-1.5	Python
Trema	proprietary	OpenFlow	1.0, 1.3	C, Ruby

Most of controllers support only OpenFlow as CDPI but some, such as OpenDaylight, Onix, and HP VAN, offer a wider range of Southbound APIs or protocol plugins. Onix supports both the OpenFlow and OVSDB protocols. The HP VAN SDN Controller has other CDPI

[5]http://www.noxrepo.org/pox/

connectors such as L2 and L3 agents. OpenDaylight provides an Service Layer Abstraction (SLA) allowing several CDPI (OpenFlow, OVSDB, NETCONF, LISP, BGP, PCEP, SNMP) to coexist in the control platform.

To communicate with the upper layers, Controllers use the NBI interface. Unlike the CDPI, there is still no strong standard adopted for the NBI, so, each controller specifies its own interface.

Northbound Interface

The Northbound Interfaces (NBI) aims to abstract the higher Application layer commands to optimized low-level CDPI instructions used to program the network elements. Usually, this interface is provided by the Controller and can be a proprietary API, a REST API or an implementation of a higher-level language. Unlike the CDPI, where OpenFlow stands out as the major initiative, in NBI there is still no standard or protocol widely used. Besides the ONF announced the creation of the Northbound Interface Working Group, in Octobre 2013, for developing NBIs information models but, currently no relevant document was released by this group [ONF, 2017].

One way to propose higher level to applications is through an API: Proprietary or REST(ful). In computer systems, client programs use APIs to communicate with servers. Generally speaking, an API exposes a set of data and functions to facilitate interactions between computer programs and allow them to exchange information [Jakl, 2008].

Proprietary APIs provide functions specified by the manufacturer, not using any development standard. In contrast, the REST API definition provides guidelines and best practices for creating scalable (web) services. Its primary goal is to allow Internet-scale distributed hypermedia systems by reducing network latency with caching and reducing server load by omitting session states. It describes some architectural constraints: client-server communication model, uniform interface, layered system, cacheable, stateless, and optionally coded on demand [Jakl, 2008].

The uniform interface standardizes four interfaces: identification of resources, manipulation of resources through representations, self-descriptive messages, and Hypermedia As The Engine Of Application State. Identification of resources is made by using Uniform Resource Identifier[6]. Manipulation of resources through representations defines the interaction with resources but not the resource itself. This conceptual distinction allows the resource to be represented in different ways and formats. The self-descriptive messages may include metadata to convey additional details regarding the resource state, the representation format and size, and the message itself. It specifies the principle that the client interacts with a network application entirely through hypermedia provided dynamically by application servers. This constraint decouples client and server to evolve independently.

The layered system constraints enable network-based intermediaries such as proxies and gateways to act transparently between a client and server. The network-based intermediary may intercept client-server communication for security enforcement, response caching, and/or load balancing. Caching constraints permit the server to declare the cacheability of each response's data. Caching response data can help to reduce latency, increase the overall availability and reliability of, and also control the server's load. Stateless constraint says that each request is an independent transaction, unrelated to any previous request. Thereby, the communication consists of independent pairs of request and response. Finally, the coded on demand constraint enables servers to temporarily transfer executable programs, such as scripts or plug-ins, to be executed

[6]Example of Uniform Resource Identifier: https://www.opennetworking.org/about/onf-overview/

by the clients. To be considered RESTful, the API must implement all these constraints. As seen in Table 2.2, just a few controllers implement the constraints specified by the REST architecture; most of them implement a proprietary API, probably due to the lack of standardization in this interface.

Another approach for simplifying the network programmability is through high-level programming languages. New languages are being proposed, increasing the level of offered abstractions, adding new features to different programming aspects, and also optimizing the utilization of network resources. Given the importance of languages for programming the network, we systematically studied them, analyzing their features, interactions, and evolutions. This state-of-the-art investigation is presented in Section 2.3.

2.1.3 Application Plane

Application Plane is the highest level in the SDN architecture. This plane is composed of one or more SDN Applications: programs that explicitly, directly, and programmatically communicate their network requirements and desired network behavior to the SDN Controller via NBIs [ONF, 2013]. There are many works proposing applications to deal with traditional network areas, such as routing, firewalling, load balancing, monitoring, QoS, and so forth.

We investigated relevant SDN applications related to this thesis, i.e., presenting an approach to optimize the network according to the requirements of applications running on top of the network. These applications are reported in Section 3.1.

Although in the architecture proposed by ONF, applications sit at the highest level of the SDN architecture, in practice, the network can be programmed through three different levels of abstraction which are discussed in the next section.

2.2 Abstraction Levels for Programming SDN

SDNs can be programmed in the three different levels: Low-level, API-based, and Domain-Specific Language (DSL) [Mernik et al., 2005]. Figure 2.4 shows the SDN architecture, pinpointing where these programming levels are located, and the following sections explain them, highlighting their differences.

The most common categories of SDN applications include forwarding and firewalling so, for explaining the differences among these programming levels, we implemented two applications: a static switch and a packet filter firewall. These examples were implemented in the three abstraction levels and deployed in a simple topology consisting of two hosts and three switches (Figure 2.5). Although simple, this topology allows illustrating some advantages in programming SDN with higher-level languages supporting fundamental concepts (constructs) such as modular composition, automatic priority handling, dynamic policy instantiation, and topology abstraction.

In a static switch, all flow tables are filled manually, using the destination MAC address to forward packets out to the physical ports. In a real environment, this example is impractical because it is not scalable, i.e. in adding a new host or switch, all flow tables must be manually reprogrammed. The example also does not allow mobility, that is, the hosts can never change the switch ports they are connected to. The final issue lies in all decisions being deployed manually, in a large environment, it becomes highly error-prone.

The packet filter firewall looks at the packet header (network addresses and ports) and determines if that packet should be allowed or blocked [Julkunen and Chow, 1998]. The second implemented application specifies packet filter rules to deny the File Transfer Protocol (FTP).

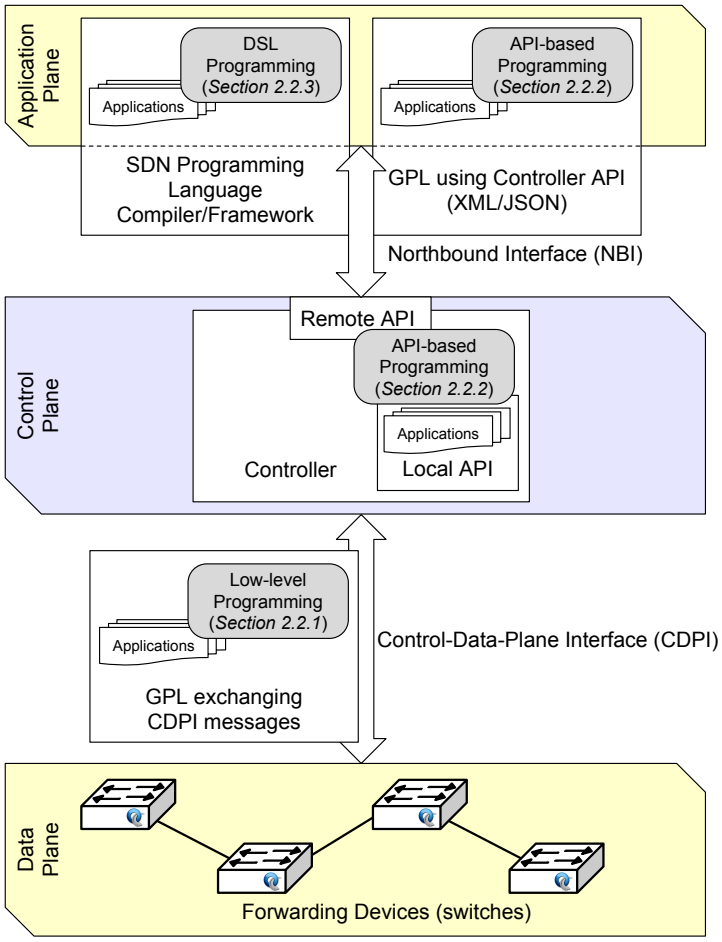


Figure 2.4: SDN programming levels.

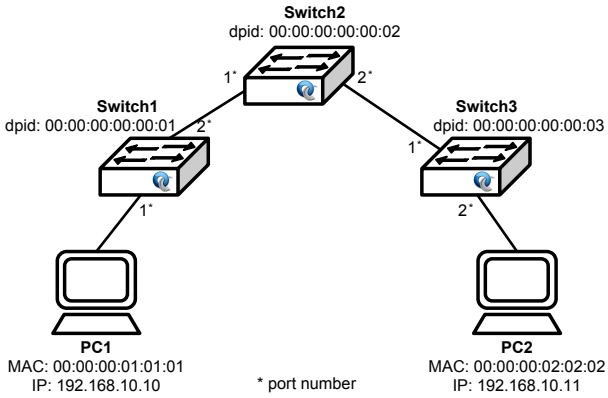


Figure 2.5: Experimental topology.

This protocol uses two TCP ports to separate control and data connections. The TCP port number 20 is used to exchange the data, and TCP port number 21 is used to transfer the protocol control messages, so both ports have to be blocked. This example demonstrates that low-level premises, such as priorities, can be abstracted through the languages' constructs.

2.2.1 Low-level

We labeled Low-level programming when the networks are programmed directly through CDPI. As we stated before, OpenFlow stands out as the major initiative in CDPI, but it does not make the programming task easier. Actually, from the perspective of programming languages, it may be considered as the assembly language of network programmability. OpenFlow forces the programmers to manipulate bit patterns in packets and carefully manage the shared rule-table space [Reich et al., 2013]. Moreover, each network device must be programmed individually, errors and inconsistencies must be manually handled by the network programmer. To develop applications in this programming level, it is necessary to use a General-purpose Language (GPL) (such as C, Java, Python, or shell script) to exchange CDPI messages directly with the switches.

To implement the first example (static switch), using Low-level programming, it is necessary to specify, in all switches, all the forwarding decisions. It is similar to configure conventional routers with static routing, but using MAC addresses rather than IP addresses. To program the switches, it has to be sent OpenFlow *FlowMod* messages, specifying the *dl_dst* in the *match* header and the destination *port* in *action* header. The structure of these messages is presented in Figure 2.6.

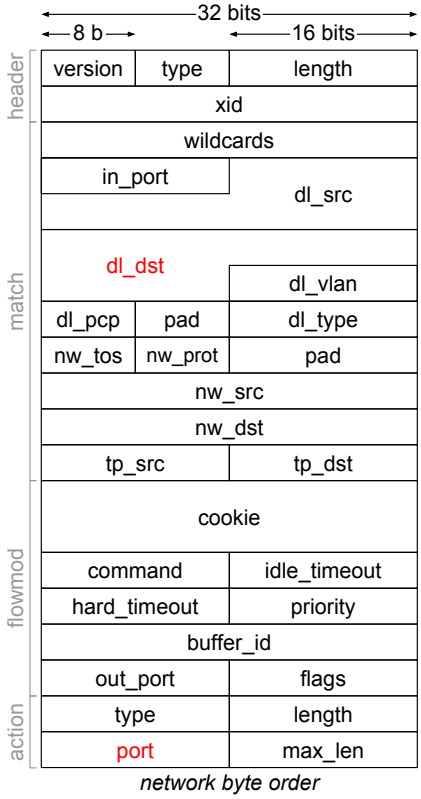


Figure 2.6: OpenFlow FlowMod message.

The topology shown in Figure 2.5 was implemented using Open vSwitch [Pfaff et al., 2009]; this software switch implementation provides a set of tools to manage the switches. The command *ovs-ofctl* was used to send the OpenFlow messages to configure the devices to act as a static switch. Figure 2.7 shows the commands, implemented in a Linux shell script, to program all switches to act as static switches.

Similarly, the second application, the packet filter firewall, was implemented in the same topology by using the command *ovs-ofctl* (Figure 2.8). In this case, it was necessary to

```
Switch1
 ovs-ofctl add-flow Switch1 priority=100,dl_dst=00:00:00:01:01:01,actions=output:1
 ovs-ofctl add-flow Switch1 priority=100,dl_dst=00:00:00:02:02:02,actions=output:2
Switch2
 ovs-ofctl add-flow Switch2 priority=100,dl_dst=00:00:00:01:01:01,actions=output:1
 ovs-ofctl add-flow Switch2 priority=100,dl_dst=00:00:00:02:02:02,actions=output:2
Switch3
 ovs-ofctl add-flow Switch3 priority=100,dl_dst=00:00:00:01:01:01,actions=output:1
 ovs-ofctl add-flow Switch3 priority=100,dl_dst=00:00:00:02:02:02,actions=output:2
```

Figure 2.7: Static switch programmed directly with OpenFlow.

insert rules into the switches to drop packets with TCP destination port equal to 20 and 21. The priority was defined to 200 to be greater than the forwarding rules presented in Figure 2.7.

```
Switch1
 ovs-ofctl add-flow Switch1 priority=200,tcp,tp_dst=20,actions=drop
 ovs-ofctl add-flow Switch1 priority=200,tcp,tp_dst=21,actions=drop
Switch3
 ovs-ofctl add-flow Switch3 priority=200,tcp,tp_dst=20,actions=drop
 ovs-ofctl add-flow Switch3 priority=200,tcp,tp_dst=21,actions=drop
```

Figure 2.8: Packet filter firewall to deny FTP ports programmed directly with OpenFlow.

In the first example, flows had to be inserted into all switches to allow the communication between PC1 and PC2. When a new host is connected to this topology, all switches have to be reprogrammed, specifying rules to correctly forward traffic to it. In the second example, it was necessary to be aware of the priority of firewalling rules to be greater than the forwarding ones; otherwise, TCP communication would not be blocked.

2.2.2 API-based

Applications implemented in this programming level use the APIs exposed by the controllers. These applications are translated by the controller into CDPI messages. However, these APIs present the same problems as Low-level programming; they oblige programmers to reason manually, in unstructured and ad hoc ways, with low-level dependencies among different parts of their code. For instance, an application that performs multiple tasks (e.g., routing, monitoring, access control, and server load balancing) must ensure that packet-processing rules installed to perform one task do not override the functionality of another. This fact leads to monolithic applications where the logic for different tasks is inexorably intertwined, making the software difficult to write, test, debug, and reuse [Monsanto et al., 2013].

The controllers may expose two types of APIs: local and remote. When using the local APIs, the SDN applications are implemented internally in the controller, using its classes, methods, and interfaces. These applications must be written in the controller's language, and must run on the same host. For example, to implement the first *ovs-ofctl* command, shown in Figure 2.7, using the Floodlight local API[7], it would be necessary the piece of code shown in Figure 2.9. The other commands used to program the static switch application have not been shown, because they are repetitive and to save space. However, the whole application can be made by repeating the Figure 2.9 code, modifying the port (line 4) and destination MAC address (line 7).

[7]Floodlight API documentation is available at:
`https://floodlight.atlassian.net/wiki/display/floodlightcontroller`

```
// Declare the flow
1. OFFlowMod fmTo=new OFFlowMod();
2. fmTo.setType(OFType.FLOW_MOD);
// Declare the action
3. List actionsTo=new ArrayList();
4. OFAction outputTo=new OFActionOutput((short)1);
5. actionsTo.add(outputTo);
// Declare the match
6. OFMatch mTo=new OFMatch();
7. mTo.setDataLayerDestination(
       Ethernet.toMACAddress("00:00:00:01:01:01"));
8. fmTo.setActions(actionsTo);
9. fmTo.setMatch(mTo);
// Push the flow
10. staticFlowEntryPusher.addFlow("FlowTo",fmTo,dp);
```

Figure 2.9: Modifying flow tables with local Floodlight API.

Remote APIs can act as an abstraction/mediation level between the two different environments; they allow the different technologies to exchange information. Usually, the information is exchanged through Extensible Markup Language (XML) or JavaScript Object Notation (JSON). Applications using this type of API can be written in any GPL and can run on a different host than the controller. We opted to implement the examples using only remote APIs because the source code is cleaner than local APIs.

POX [McCauley, 2015] is a popular controller that provides a remote API[8] for modifying the switches' flow tables. Figure 2.10 shows the required JSONs to specify the static forwarding. The program *curl* may be used for sending these JSON to the controller. The API *set_table* method was used to insert the flows into switches' flow tables.

```
Switch1
 '{"method":"set_table","params":{"dpid": "00-00-00-00-00-01","flows":
  [{"actions":[{"type":"OFPAT_OUTPUT","port":1}],
    "match":{"dl_dst":"00:00:00:01:01:01"}},
  {"actions":[{"type":"OFPAT_OUTPUT","port":2}],
    "match":{"dl_dst":"00:00:00:02:02:02"}}]}}'
Switch2
 '{"method":"set_table","params":{"dpid": "00-00-00-00-00-02","flows":
  [{"actions":[{"type":"OFPAT_OUTPUT","port":1}],
    "match":{"dl_dst":"00:00:00:01:01:01"}},
  {"actions":[{"type":"OFPAT_OUTPUT","port":2}],
    "match":{"dl_dst":"00:00:00:02:02:02"}}]}}'
Switch3
 '{"method":"set_table","params":{"dpid": "00-00-00-00-00-03","flows":
  [{"actions":[{"type":"OFPAT_OUTPUT","port":1}],
    "match":{"dl_dst":"00:00:00:01:01:01"}},
  {"actions":[{"type":"OFPAT_OUTPUT","port":2}],
    "match":{"dl_dst":"00:00:00:02:02:02"}}]}}'
```

Figure 2.10: Static switch using the POX controller API.

[8]POX API documentation is available at:
`https://openflow.stanford.edu/display/ONL/POX+Wiki`

Some controllers offer specific APIs to ease the development of network applications; the Ryu controller, for example, provides APIs to deal with firewalling, routing, and setting QoS. So, to implement the second application, we used the Ryu Firewall API[9].

Figure 2.11 shows the necessary JSONs to be sent to switches to deny the FTP ports. Due to the OpenFlow matching fields prerequisites, matching on the TCP port is allowed only if the Ethernet type is explicitly set to IPv4 and the IP protocol is set to TCP. In the same application, implemented with Low-level programming (Figure 2.8), the *ovs-ofctl* command includes a shorthand notation to specify the Ethernet type and IP protocol[10].

```
Switch1
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "20",
   "actions": "DENY", "priority": "200"}'
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "21",
   "actions": "DENY", "priority": "200"}'
Switch3
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "20",
   "actions": "DENY", "priority": "200"}'
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "21",
   "actions": "DENY", "priority": "200"}'
```

Figure 2.11: Packet filter firewall to deny FTP ports using the Ryu Firewall API.

Controllers provide APIs to program the network but, as it was possible to see through these examples, the functionalities offered are low-level, just like the OpenFlow protocol. We could also see that, even using a specific firewall API, all rules must be inserted manually in the right switches, and the rules priority must be carefully managed to avoid forwarding rules with higher priority.

In these two first programming levels, to create an application with multiple functionalities, for example, the static switch combined with the firewall, it is necessary to create a unique program implementing the both functions. Again, the rules priorities must be carefully inserted to avoid higher priority functions from being overlapped.

2.2.3 Domain-Specific Language

A Domain-Specific Language (DSL) is a programming language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [Van Deursen et al., 2000]. The SDN programming languages have a compiler (or framework) responsible for translating their high-level constructs to messages understandable by a controller API. All the studied SDN programming languages were classified as Domain-Specific Language (DSL), providing higher-level constructs to optimize the network resources utilization, and adding new abstractions for different programming aspects, such as monitoring, security, and virtualization.

Some languages were designed to address specific problems, while others offer more general abstractions, and can be used to create any network application. To emphasize the differences in abstractions offered by the different SDN languages, we implemented the first application, the static switch, in two programming languages: Pyretic [Reich et al., 2013, Monsanto et al., 2013] and Merlin [Soulé et al., 2013a].

[9]Ryu Firewall API documentation is available at:
`https://osrg.github.io/ryu-book/en/html/rest_firewall.html`

[10]The *ovs-ofctl* command translates the *tcp* keyword to "dl_type=0x0800, nw_proto=6"

Figure 2.12 shows the static switch implemented in Pyretic[11]. The Pyretic's compiler translates network applications into Ryu (or NOX) remote API commands. The Pyretic language is embedded in Python, and every program must have a *main* method and import at minimum the Pyretic core library.

To perform the static switch behavior, we defined a method called *static_switch*. This method uses the construct *match* to filter packets with destination MAC address 00:00:00:01:01:01 (PC1). The forwarding to port 1 is performed through construct *fwd*. Similarly, the packets to PC2 are forwarded to port 2. This was only possible because all switches use port 1 to forward to PC1 and port 2 to forward to PC2; usually, that is not true, so it is necessary to include a new match to specify, in each switch, which port should be forwarded (e.g. $((match(switch = Switch1)\ \&\ match(dstmac = EthAddr(\text{'00:00:00:01:01:01'}))) >> fwd(1))$).

```
from pyretic.lib.corelib import *

def static_switch():
  return ((match(dstmac=EthAddr('00:00:00:01:01:01')) >> fwd(1)) +
          (match(dstmac=EthAddr('00:00:00:02:02:02')) >> fwd(2)))

def main():
  return static_switch()
```

Figure 2.12: Static switch programmed with Pyretic.

We also implemented the static switch in the Merlin framework[12]. This framework that allows administrators to express network policies in a high-level, declarative language (Section 2.3.3) based on regular expressions. Figure 2.13 shows the source code of a static switch implemented in Merlin.

```
ipSrc = 192.168.10.10 and ipDst = 192.168.10.11 -> .* ;
ipSrc = 192.168.10.11 and ipDst = 192.168.10.10 -> .* ;
```

Figure 2.13: Static switch implemented in Merlin.

The Merlin language provides a higher-level abstraction ($.*$) to specify path selection; the dot symbol indicates matches any single network device and the asterisk means repetition. This abstraction can be used to define the forwarding without worrying about the switches, links, or ports; it automatically installs the necessary rules in the right switches. The Merlin compiler computes the shortest path between the hosts and programs the flow entries in switches between PC1 and PC2. Further details on path selection abstraction are presented in Section 2.3.4.

The second example, the packet filter firewall to deny FTP ports, was also implemented in Pyretic and its source code is presented in Figure 2.14. To implement this new function, we created another method, called *firewall*, to drop all packets matching TCP ports 20 and 21. The construct negation ($\sim$) in conjunction with *match*, filter only packets not matching the specified destination TCP ports (*dstport*), so FTP packets will be dropped, and all other packets can be forwarded.

[11]Pyretic documentation is available at:
`https://github.com/frenetic-lang/pyretic/wiki/Language-Basics`

[12]Merlin documentation is available at:
`https://github.com/merlin-lang/merlin`

```
from pyretic.lib.corelib import *

def firewall():
  return (~match(dstport=20) & ~match(dstport=21))

def static_switch():
  return ((match(dstmac=EthAddr('00:00:00:01:01:01')) >> fwd(1)) +
          (match(dstmac=EthAddr('00:00:00:02:02:02')) >> fwd(2)))

def main():
  return firewall() >> static_switch()
```

Figure 2.14: Packet filter firewall to deny FTP ports programmed with Pyretic.

Both functions static switch and packet filter firewall are finally composed in the *main* method through the sequential modular compositor (>>). This compositor receives the packets outputted from *firewall* method (all packets except those with TCP destination ports 20 and 21), and forwards these packets to the *static_switch* method. The forwarding decisions implemented in the *static_switch* method are then performed.

Despite the simplicity of the examples implemented in a tiny topology, we can observe a substantial difference in favor of the languages, regarding topology abstraction, automatic priority control, or dynamic policy instantiation. Topology abstraction is relevant especially when the topology complexity increases. Automatic priority control, dynamic policy instantiation, and modular composition (Section 2.3.4) are useful to overcome the intricacy of multiple concurrent applications running on the network. Moreover, as stated earlier, in the first two programming levels, all rules must be inserted individually in every switch, where the different tasks priorities have to be handled manually.

The abstractions presented in these examples are not exhaustive concerning those offered by SDN languages, on the contrary, languages are evolving and increasingly providing higher-level constructs. Due to the wide variety of abstractions that languages can offer, we surveyed the existing SDN languages [Trois et al., 2016b], reporting and classifying their abstractions.

2.3 SDN Programming Language Classification

We systematically studied the languages, proposing a taxonomy for classifying them. First, we delimited, among the existing NBI publications, those proposing a language to program the network. Next, we gathered all the prominent language features, mapping and grouping the similar ones. This step was difficult because, as we stated before, there is no NBI standardization, and many different nomenclatures were used for naming similar features. All surveyed languages were classified as DSL [Mernik et al., 2005]. Some of these languages provide a general and wide set of operations, including abstractions to query network information, to specify flow matching and actions, to compose programs by grouping modules, and so forth. On the other hand, some languages yield to specific areas, such as fault-tolerance or policy delegation.

For elaborating the taxonomy, we used the Feature-Oriented Domain Analysis (FODA) [Kang et al., 1990] method. The primary focus of this method is the identification of the prominent or distinctive features on specific domains, in our case, the SDN programming languages. Features are the attributes of a system that directly affect end users. The end users have to make decisions regarding the availability of features in the system, and they need to understand the meaning of the features in order to use the system. A feature diagram is a hierar-

chically arranged set of features. Relationships between a parent feature and its child features are categorized as: *and* – all sub-features must be selected, *alternative* – only one subfeature can be selected, *inclusive or* – one or more can be selected, *mandatory* – features that required, and *optional* – features that are optional [Batory, 2005]. Figure 2.15 presents the symbology to represent the feature diagrams.

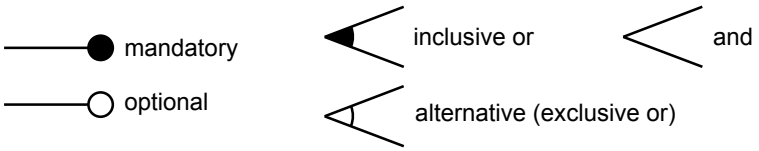


Figure 2.15: Feature diagram legend.

We firstly researched papers proposing NBI abstractions. Once those papers were identified, we used the FODA Context Analysis to filter only the proposals presenting language-related aspects as key characteristics. These papers also described other information such as language framework, compiling techniques and optimizations, and so forth – we focused only on language aspects. In the next FODA phase (Domain Modeling), all relevant features and their relationships were raised and mapped into a feature diagram. Due to the lack of standardization, some features presented with various names by different authors were grouped.

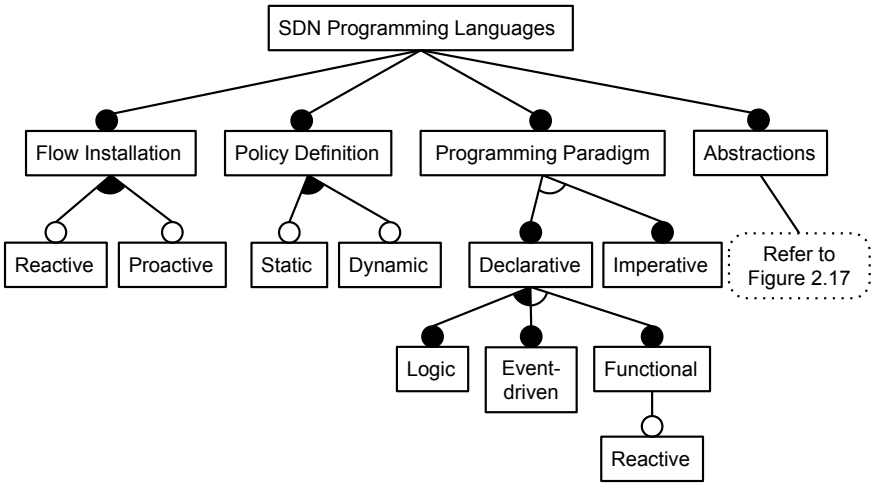


Figure 2.16: Top-level classification feature diagram.

Figure 2.16 shows the top-level feature diagram, where the first subnode level represents the major axes: Flow Installation, Policy Definition, Programming Paradigm, and the Abstractions provided by the languages. Table 2.3 classifies the languages according to these features. The following sections explain these features in further details. For better visualization, the *Abstractions* were plotted in a separate feature diagram (Figure 2.17). We did not include in our taxonomy elementary OpenFlow operations such as forwarding, packet header modification, and others. The features were identified based on the existing SDN programming languages. They might not be exhaustive with respect to all SDN requirements, and additional features may be included with the evolution of the languages.

2.3.1 Flow Installation

Flow installation refers to the way the languages install the rules on switches. An OpenFlow switch forwards packets according to rules stored in its flow tables. The controller

handles these tables by adding or removing those rules. The rules can be installed in switches either reactively (when a new flow arrives on the switch), proactively (controller installs rules beforehand), or in a hybrid way. Hybrid flow installation is a combination of both, *reactive* and *proactive* approaches.

In the *reactive* approach, when a new packet arrives at the switch, a lookup operation is performed in its flow tables. Then if there is no match between the arriving packet and switch's flow table entries, the switch creates an OpenFlow *packet-in* packet and sends it to the controller. The controller receives the *packet-in* and, based on its program logic, it creates and installs a rule in the switch's flow tables. Languages that use reactive flow installation forward all new flows to controllers. Among the surveyed languages, we observed several languages using reactive flow installation [Hinrichs et al., 2009, Voellmy et al., 2010, Foster et al., 2010, Voellmy et al., 2012, Katta et al., 2012, Soulé et al., 2014, Batista and Fernandez, 2014, Trois et al., 2015]. The main problem with this reactive approach is that the latency is increased because the controller must be consulted on every new flow.

Proactive installation eliminates the latency induced by consulting a controller on every new flow. The flow tables are populated ahead of time for all traffic matches that could come into the switch. By pre-defining all flows and actions in the switches' flow tables, the *packet-in* event never occurs. As the result, all packets are forwarded at switch processing capacity (e.g. line rate), by doing a flow lookup in the switches' flow tables [Salisbury, 2013]. Languages that provide proactive installation pre-compute forwarding tables for the entire network, and the controller just modifies the switches' flow tables when it has to react in case of link failures or any other external event. The proactive approach is harder to be implemented because it requires the controller to know the traffic flows in advance[13], usually by keeping the traffic history, to configure all paths before they are used [Fernandez, 2013]. We found languages providing proactive flow installation natively [Monsanto et al., 2012, Foster et al., 2013, Monsanto et al., 2013, Anderson et al., 2014] and also languages that were implemented on top of these, inheriting the proactivity [Nelson et al., 2013, Reitblatt et al., 2013, Kim et al., 2015]. All these languages also implement the reactive aproach.

2.3.2 Policy Definition

A policy is formally defined as "an aggregation of policy rules" [Stone et al., 2001]. Each policy rule is comprised of a set of conditions and a corresponding set of actions. The conditions define when the policy rule is applicable. Once a policy rule is activated, one or more actions contained by that policy rule may be executed. SDN programming languages provide high-level statements to define network policies. These policies can be triggered in two ways, statically or dynamically. We classified the network languages according to this aspect. It is important to realize that a language can specify static policies, dynamic policies or both.

Static policies apply a fixed set of actions in a pre-determined way according to a set of pre-defined parameters [Stone et al., 2001]. Static policies are the most traditional method of firewalling. An example of a static rule-based policy would be to allow/deny an IP address to access a server. Other examples of static policies: social network traffic in not allowed during regular working hours; video-sharing websites are only permitted after 5:00 PM.

FML [Hinrichs et al., 2009], the first SDN programming language, is the only one that allows only static policies. The code below is one FML example using static policy to allow flows from TCP port 80 and IP source 10.0.0.1.

[13] wildcard rules can be used for flow agregation

Table 2.3: Languages features summarization.

Feature \ Language	FML	Nettle	Frenetic	Procera	Flog	NetCore	Frenetic-Ocaml	Pyretic	FlowLog	FatTire	NetKAT	Merlin	PonderFlow	NoF	Kinetic
Flow Installation	R	R	R	R	R	R/P	R/P	R/P	R/P*	R/P*	R/P	R	R	R	R/P*
Policy Definition	S	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Programming Paradigm	L	FRP	FRP	FRP	L/ED	FRP	FRP	I	v1:L v2:ED	F	F	F	ED	ED	ED

Legend

* = inherited	R=Reactive, P=Proactive	S=Static D=Dynamic and Static	I=Imperative, L=Logic, ED=Event-driven F=Functional, FRP=Funct. Reactive

$$allow(Flow) \Leftarrow Prot = 80 \wedge H_s = 10.0.0.1$$

Unlike the static, *dynamic* policies are enforced based on changing conditions of the network, such as congestion, packet loss, bandwidth usage, the loss of a network router, or any external event. To support the dynamic, and sometimes the unexpected nature of the network, actions can be triggered when an event causes a policy condition to be met [Stone et al., 2001]. Many network systems have dynamic behavior, for example, intrusion detection and prevention systems detect certain sequences of events and trigger action to deny the source of these events. Another example is a load balancing system that uses server load and other external events to change the network behavior.

Network administrators may want to limit the network access to devices and users based on maximum bandwidth usage. Suppose one want to limit the bandwidth on 5GB/min for all incoming HTTP traffic to a pair of hosts *h1* and *h2*. It is possible to express these constraints by using the following policy:

$$\begin{array}{l}(bytes \\ \quad \{\, location\ \{\, h1, h2\ \}\ and\ tcp.src\ \tilde{}\ 80\ over \\ \qquad tumbling(60)\ \} \\ )\ <\ 5GB\end{array}$$

This policy is written in Merlin [Soulé et al., 2013b] and it is made up of several nested constructs. The inner-most construct describes a set of packets: all packets on hosts *h1* and *h2* with TCP source port 80 sent or received in the last 60 second tumbling window. Next, the aggregate function *bytes* reduces this set to a scalar value by taking the sum of the sizes of all packets in the set. Finally, the logical constraint stipulates that this scalar must be less than 5GB. Except FML [Hinrichs et al., 2009], all the other languages allow to create dynamic policies.

2.3.3 Programming Paradigm

The programming paradigm differs the way of building the structure and elements of the computer (or network) programs. Capabilities and styles of programming languages are defined by their supported programming paradigms. Each paradigm contains a set of concepts that make it more appropriate to solve certain kind of problems. For example, object-oriented programming is more suited for applications with a large number of related data abstractions, often organized in a hierarchy. Logic programming is well suited for transforming or navigating

complex symbolic structures according to logical rules [Van Roy et al., 2009]. Next sections explain the paradigms found in SDN languages.

Imperative

The imperative paradigm can be viewed as the traditional computer programming model. This paradigm lets the programmer specify all the steps to solve a particular task. Computation is seen as a task to be performed by a processing unit that manipulates and modifies memory. Imperative languages provide controls to modify execution flow: loops, conditional branching statements and procedures [Mottola, 2005].

The well-known imperative languages are FORTRAN, COBOL, Basic, Pascal, C, C++, PHP, Python, Java. Among the surveyed languages, only Pyretic [Monsanto et al., 2013] uses this paradigm. The next example is a Pyretic function to add rules to a firewall:

```
def AddRule (self, mac1, mac2) :
  if (mac2, mac1) in self.firewall :
    print "Firewall rule already exists"
    return
  self.firewall[(mac1,mac2)] = True
  print "Adding rule in %s : %s" % (mac1,mac2)
  self.update_policy()
```

The program verifies *if* the pair of MAC addresses *mac1* and *mac2* are set to *True* in the *self.firewall* list. If so, it just *prints* a message on the terminal; otherwise it sets to *True* in the list. As can be seen in this example, the language provides the traditional *if* and *return* conditional statements and also function and variables definitions.

Declarative

In declarative programming, one can specify what the program must do, without specifying how to do it [Qadir and Hasan, 2015] [Sharan, 2012]. Probably the most known example of declarative languages is the Structured Query Language (SQL) used to query computer databases, where a query is stated and the database engine is responsible for executing it. As sub-paradigm of these declarative languages, we found proposals fitting in *logic*, *event-driven*, *functional*, and also *functional reactive* paradigms, or a mix.

Logic programming is a declarative paradigm based on formal logic. The language compiler applies an algorithm that scans every possible combination into a set of defined inference rules applied to the axioms to resolve queries [Lloyd, 2012]. The most known examples are Prolog [Clocksin and Mellish, 2003], ASP [Lifschitz, 2008], and Datalog [Li and Mitchell, 2003]. Logical languages may implement recursion and negation. Recursion indicates that the algorithms can be implemented by means of recursive predicates. Negation allows users to specify negative predicates. We perceived three languages defined as logic, based on Datalog. FML [Hinrichs et al., 2009], FlowLog v_1 [Nelson et al., 2013], and Flog [Katta et al., 2012] that uses both logic and event-driven paradigms.

Event-driven programming enables the programs to respond to events. Upon receiving an event, an action is automatically triggered. The action may carry out some computation as well as give rise to some new event [Ucoluk and Kalkan, 2012]. Generally in event-driven paradigm, there exists a hidden main loop listening to the events and dispatching the appropriate actions. The following example is a program written in the second version of FlowLog [Nelson et al., 2014] to blacklist all IP address trying to use telnet protocol (TCP port 23). When a *packet_in* event is received, this code is executed. The program filters all packets with TCP port = 23, and then add the sender's IP address to a blacklist.

```
ON packet_in(p) WHERE p.nwProtocol = 23 :
  INSERT (p.nwSrc) INTO blackList ;
```

Many event-driven applications are stateless: when the application finishes processing an event, the application was not changed by it. The opposite of a stateless application is a stateful application. Stateful are applications modified by their processed events. Specifically, stateful applications remember or maintain information between events, and what they remember can be changed by events [Ferg, 2006]. Flog [Katta et al., 2012] is a mix between logic and event-driven. The second version of FlowLog [Nelson et al., 2014], PonderFlow [Batista and Fernandez, 2014], NoF [Trois et al., 2015], and Kinetic [Kim et al., 2015] were considered event-driven languages.

Functional programming treats computation as the evaluation of mathematical functions, avoiding state changes and mutable data. Functional programming languages become stateless and provide referential transparency, meaning that the result of a function will be the same for a given set of parameters no matter where, or when, it is evaluated. Referential transparency greatly enhances modularity, as a function does not depend on the context where it is being executed [Mottola, 2005]. Thanks to the brevity, expressiveness, and availability of sophisticated data structures and algorithms, modern functional programming languages are being used in a wide range of scientific applications, from numerical analysis to visualization [Mottola, 2005]. As examples of functional computer programming languages, one can cite Erlang [Armstrong, 2013], OCaml [Rémy, 2002], Clojure [Halloway, 2009], and Haskell [Thompson, 1999]. Among the surveyed languages, we encountered three functional languages [Reitblatt et al., 2013, Anderson et al., 2014, Soulé et al., 2013a].

The *reactive* programming paradigm provides abstractions to express programs as reactions to external events. It automatically manages time and data flows and also computation dependencies. Programmers do not need to worry about the order of events. This property of automatic management of data dependencies can be observed in spreadsheet systems. A spreadsheet typically consists of cells, which contain values or formulas. If a value of a cell changes then the formulas are automatically recalculated [Bainomugisha et al., 2012].

Functional Reactive Programming (FRP) [Wan and Hudak, 2000] is a method of modeling reactive behavior in purely functional languages. FRP permits the modeling of systems that must respond to input over time in a simple and declarative manner. A program in an FRP language corresponds quite closely to a mathematical model of the system being implemented. The primary concepts of FRP are signals (time-varying values) and events (collections of instantaneous values, or time-value pairs). FRP achieves reactivity by providing constructs for specifying how signals change in response to events [Amsden, 2011]. The following code is a simple Nettle [Voellmy et al., 2010] example to flood all network packets. The *packetIns* filters only the OpenFlow *PacketIn*[14] messages from the incoming message stream. For each such event, it applies *sendRcvdPacket flood*, instructing the switch to send the port using the action *flood*, which results in the switch forwarding the packet on every port except the incoming port.

```
floodPackets1 = proc evt → do
  packetInEvt ← packetIns ≺ evt
    returnA ≺ sendRcvdPacket flood packetInEvt
```

Elm [Czaplicki and Chong, 2013], Flapjax [Meyerovich et al., 2009], and Yampa [Courtney et al., 2003] are examples of FRP languages for computer program-

[14]OpenFlow PacketIn message is sent from switch to the controller, when there is no match on switch's flow tables.

ming. FRP was used in five studied languages [Voellmy et al., 2010, Foster et al., 2010, Voellmy et al., 2012, Monsanto et al., 2012, Foster et al., 2013].

2.3.4 Abstractions

An abstraction is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the user and suppresses details that are, at least for the moment, immaterial or diversionary [Shaw, 1984]. Abstractions provide a good way to manage complexity [Szyperski et al., 1999]. The abstractions used in programming languages tend to emphasize the functional properties of the software – what is computed rather than how the computation is carried out.

In a recent work, Casado et al. [Casado et al., 2014] wrote about overall SDN abstractions splitting them on the following areas: (1) network-wide structures provide abstractions to discover the topology, link fault detection, information about host locations, link capacities, and others. (2) Distributed updates propose abstractions with mechanisms that provide consistency guarantees during updates in network devices. (3) Modular composition allows programs to be decomposed into several modules. It also provides compositional functions (e.g., parallel or serial) to group these modules. (4) Virtualization decouples the application logic from the physical topology. This abstraction simplifies programs, ensures isolation, enhances scalability, and provides portability and fault-tolerance. (5) Formal verification provides tools for automatically checking formal properties and diagnosing user's network application problems. The application of formal methods in networking was exhaustively studied by Qadir and Hasan [Qadir and Hasan, 2015]. In their survey, they present several studies focusing on formal verification, including programming languages, data plane and control plane verification, and also debugging tools.

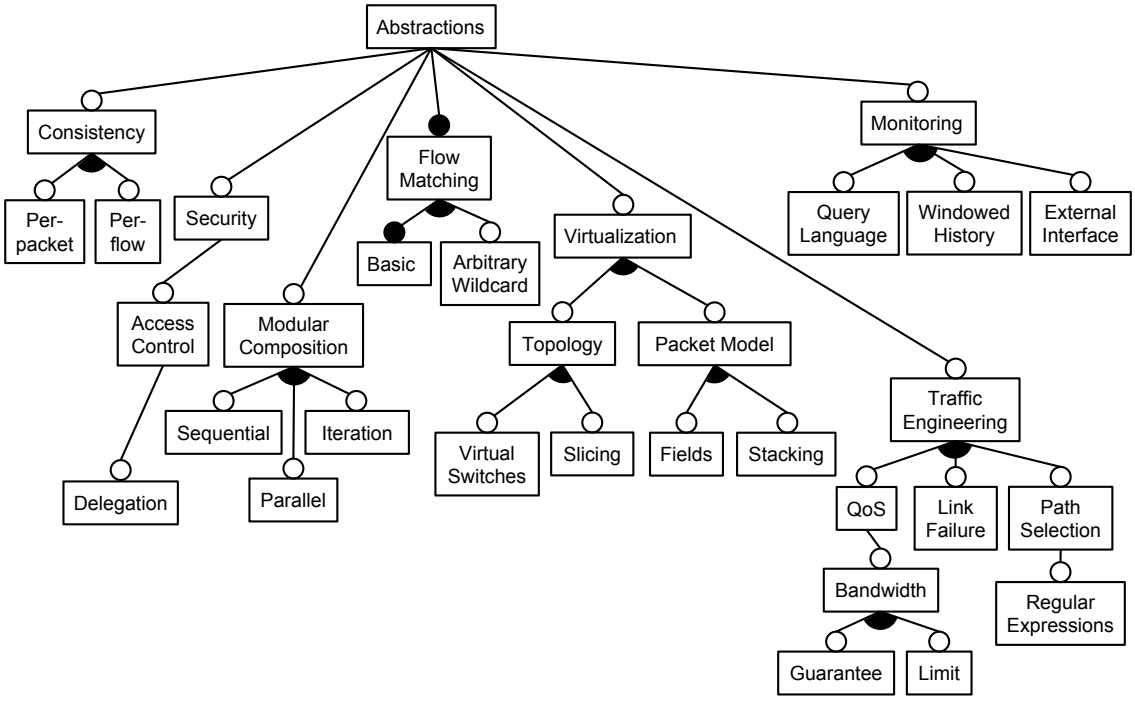


Figure 2.17: Language abstractions feature diagram.

Some abstractions raised by Casado et al. were implemented in the studied languages. Figure 2.17 shows the abstractions reported in the studied languages while Table 2.4 summarizes the languages supporting these abstractions. The next sections briefly explain these abstractions, citing the languages that implement them.

Table 2.4: Languages abstractions summarization.

Feature			Language	FML	Nettle	Frenetic	Procera	Flog	NetCore	Frenetic-Ocaml	Pyretic	FlowLog	FatTire	NetKAT	Merlin	PonderFlow	NoF	Kinetic
Consistency										P/F								
Security	Access Control														X	X		
		Delegation													X			
Modular Composition					S/P	S	S/P		S	S/P	S/P	S/P*	S/P*	S/P/I				S/P*
Flow Matching	Basic			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X*
	Arbitrary Wildcard					X			X	X	X			X				X*
Virtualization	Topology	Virtual Switches									X			X				
		Slicing									X			X	X		X	
	Packet Model	Fields									X							
		Stacking									X							
Traffic Engineering	QoS			L											L/G			L
	Link Failure												X					
	Path Selection			L									RE	RE	RE			
Monitoring	Query Language					X		X		X		X						
	Windowed History					X	X		X	X	X			X	X			X*
	External Interface			X			X		X			X					X	X

Legend

X = Natively * = Inherited	P=Per-packet F=Per-flow	S=Sequential, P=Parallel, I=Iteraction	L=Limit G=Guarantee	L=Language Construct, RE=Regular Expression

Consistency

This abstraction enables a controller to update the forwarding state of the entire network, ensuring a packet never traverse a path during a transition between two states. Every packet is either processed with the old configuration, or the new network configuration, but not a mixture of the two [Casado et al., 2014]. With consistency abstraction, programmers can bypass the problem of verifying program invariants across network updates. They just need to verify the initial and final states [Reitblatt et al., 2011].

There are works focusing on consistency abstractions [Ghorbani and Caesar, 2012, Katta et al., 2013, Mahajan and Wattenhofer, 2013, Noyes et al., 2014] but, the consistency is performed automatically through mechanisms implemented in these solutions. Reitblatt et al. [Reitblatt et al., 2011, Reitblatt et al., 2012] extended Frenetic [Foster et al., 2011], including abstractions to support *per-packet* and *per-flow* consistency. A per packet consistent update guarantees that every packet flowing through the network is processed with exactly one forwarding policy. Analogously, per flow consistency guarantees that all packets in the same flow

are handled by the same version of the policy. Frenetic-OCaml [Foster et al., 2013] maintained support to this abstraction.

Security

All surveyed languages present some basic security aspects for denying or allowing specific traffic; these basic aspects were not included in the taxonomy. The existing research can mostly be classified into two main categories: network access control and attack detection and defense [Suzuki et al., 2014]. Attack detection and defense is usually handled through specific SDN applications [Feng et al., 2009, Braga et al., 2010, Giotis et al., 2014] or frameworks [Wang et al., 2012b, Shin et al., 2013, Zaalouk et al., 2014]. Network *access control* can be implemented through specific solutions [Klaedtke et al., 2014, Matias et al., 2014], but also can be done via programming language constructs.

Access control is the function of specifying access rights to resources, which is related to information security and computer security. More formally, "to authorize" is to define an access policy [Liu and Xu, 2012]. The access control abstractions allow to define what services or resources an user can access. For example, in a multi-tenancy data center, it is possible to define if a user has permission to add or remove a flow in a specific switch. Authorization is useful to allow tenants to modify global network policies to suit their own demands. Among the researched languages, we reported Merlin [Soulé et al., 2013b] and PonderFlow [Batista and Fernandez, 2014] providing authorization. Merlin also allows to *delegate* the authorizations hierarchically.

```
inst auth + switchPolicyOps {
  subject < User > /NetworkAdmin;
  target < OFSwitch > /switches;
  action addFlow(), removeFlow(), enable(), disable();
}
```

The previous example shows a PonderFlow [Batista and Fernandez, 2014] program for creating a positive authorization policy allowing all network administrators to add/remove flows and to enable/disable all switches in the network.

Modular Composition

Modularity is a useful technique for controlling the complexity of programs. Programs are decomposed into separated modules with precisely specified and tightly controlled interactions. Separate modules are interesting for maintainability and code reuse. By using modularity, it is necessary to specify pathways for interaction among the modules to form a complete system [Mitchell, 1996]. Building modular SDN applications requires support for composition of multiple independent modules where each module partially specify how traffic should be handled [Monsanto et al., 2013]. Modules can be understood as packet-processing functions that consume a packet history and produce a set of packet histories [Smolka et al., 2015]. For grouping these modules, we identified three different modular compositors: *sequential*, *parallel*, and *iteration*.

Sequential composition gives the illusion of one module operating on the packets produced by another. Given two policy functions f and g operating on a located packet p, sequential composition applies g to each of the located packets produced by $f(p)$, to produce a new set of packets [Monsanto et al., 2013]. For example, suppose the programmer has one firewall module, and another to route packets inside his network. These modules can be grouped with a sequential compositor so that the received packets will be initially processed by the firewall

policies, and then be internally routed in the network. Seven languages presented native support to sequential composition [Voellmy et al., 2010, Foster et al., 2010, Voellmy et al., 2012, Monsanto et al., 2012, Foster et al., 2013, Monsanto et al., 2013, Anderson et al., 2014] and three inherited it [Nelson et al., 2013, Reitblatt et al., 2013, Kim et al., 2015].

Parallel composition enable multiple policies operating concurrently on separate copies of the same packets. Given two policy functions f and g operating on a located packet p, parallel composition computes the multiset union of $f(p)$ and $g(p)$ – that is, every located packet produced by either policy [Monsanto et al., 2013]. For example, one module monitors traffic by source IP address, while another one routes traffic by destination IP address. By using the parallel compositor, both route and monitor functions are performed simultaneously. Among the surveyed languages, this compositor was found natively in [Voellmy et al., 2010, Voellmy et al., 2012, Foster et al., 2013, Monsanto et al., 2013, Anderson et al., 2014], and it was inherited by [Nelson et al., 2013, Reitblatt et al., 2013, Kim et al., 2015].

The *iteration* composition was found only in NetKAT [Smolka et al., 2015]. This operator allows a function f to behave as the union of f composed with itself zero or more times. Once declarative languages do not implement loop statements, an iteration operator may be useful in some cases the function must be executed recursively, for example, a routing function.

Flow Matching

OpenFlow switches have flow tables, and these tables contain a set of flow entries. Each flow entry consists of match fields, counters, and a set of actions. Packet match fields are used for table lookups, and typically they include various packet header fields, such as TCP port, Ethernet source, or IP destination address. On receipt of a packet, an OpenFlow switch starts by performing a lookup in its flow tables. A packet matches a flow table entry if the packet header fields match those defined in the flow table entry [ONF, 2014]. SDN programming languages must provide matching abstractions; they were classified into two categories: *basic* matching and *arbitrary wildcard*.

A language is classified as *basic* matching when it supports the standard matching operations specified by OpenFlow. OpenFlow v1.0 provides specification to support exact match and prefix wildcard. In an exact match, the values on packet headers must be identical to those values defined in the match table. A match field may be wildcarded by its prefix (e.g., $192.168.*.*$). In newer versions of OpenFlow (v1.1+), the switches may optionally support arbitrary wildcards by using bitmask (e.g., $192.*.10.*$). The bitmasks are defined so that a 0 in a given bit position indicates a "don't care" match for the same bit in the corresponding field, whereas a 1 means match the bit exactly [ONF, 2014]. Some proposed languages were classified as basic matching [Hinrichs et al., 2009, Voellmy et al., 2010, Voellmy et al., 2012, Katta et al., 2012, Nelson et al., 2013, Reitblatt et al., 2013, Soulé et al., 2013a, Batista and Fernandez, 2014, Trois et al., 2015] because they implement only the matching abstractions offered by OpenFlow.

OpenFlow v1.0 compliant switches do not support arbitrary wildcards in matching operations. Some languages present abstraction to *arbitrary wildcard* [Foster et al., 2010, Monsanto et al., 2012, Foster et al., 2013, Monsanto et al., 2013, Anderson et al., 2014, Kim et al., 2015], even if the hardware does not support it. If the switch has basic support for wildcards, then the compiler needs to generate a policy that may be larger than would be practical. For example, this wildcard $192.*.10.*$ should be converted in 256 flow entries from $192.0.10.*$ until $192.255.10.*$. Thus, the compiler generates overapproximations

that match a superset (192. ∗ . ∗ .∗) of the traffic specified by the original predicate and uses the reactive strategy (Section 2.3.1) to send these packets to the controller [Monsanto et al., 2012].

Virtualization

This abstraction refers to the act of decoupling the service (logical) from its realization (physical) [Casado et al., 2010]. In computer science, virtualization was used since the mainframes era. The concept of multiple coexisting logical networks has already appeared in networking, presented as Virtual LANs (VLANs), Virtual Private Networks (VPNs), Active Networks [Tennenhouse et al., 1997], and overlay networks [Chowdhury and Boutaba, 2010].

According to Jain and Paul [Jain and Paul, 2013], virtualization is useful to provide: (1) Sharing – when a resource is overestimated for a single user, then it might be better to divide it into multiple virtual ones. (2) Aggregation – if the available resources are small, it is possible to create a larger virtual resource by grouping the small ones. (3) Isolation – in a shared network, enables to provide virtual "isolated slices" for the different users, preventing their information from being accessed. (4) Dynamics – enables the resources to be easier relocated, and (5) ease of management – virtual devices are easier to manage because they are software-based and expose a uniform interface through standard abstractions. With the advent of SDN, this concept grew widely, and it is being used to virtualize many network elements such as ports, cards, switches, links, and functions (NFV) [Batalle et al., 2013]. It is also used to virtualize topologies by mixing virtual and physical devices in an abstract topology. In our research, we organized these abstractions offered by languages into two groups: *topology* and *packet model*.

Network virtualization allows multiple virtual networks to run over a shared infrastructure, and each virtual network can have a much simpler (more abstract) topology than the underlying physical network [Feamster et al., 2013]. The abstract *topology* may consist of a mix of physical and virtual switches, and may have multiple levels of nesting on top of the real network [Monsanto et al., 2013]. Existing languages proposes two ways to virtualize the topologies. The first topology abstraction, also referred as one-to-many or multiplexing [Nascimento et al., 2011], allows the programmers to create multiple *virtual switches* in the same physical device. It usually is accomplished through software switches [Pfaff et al., 2009, Fernandes and Rothenberg, 2014, Floodlight, 2017], but this abstraction is also available in two languages [Monsanto et al., 2013, Smolka et al., 2015].

The second topology abstraction was named *slicing*. It has different designations in the studied papers: big switch, aggregation, overlay networks, many-to-one, and slicing. In our taxonomy, we grouped these concepts, understanding that there is no significant difference between them to justify such separation. This abstraction allows the network to be "sliced". A slice is composed by a subset of switches, ports, and links. The packet processing on each slice is dictated exclusively by the program for that slice, not being affected by the programs for any other slices (isolation). By slicing a network, it is possible to define which packets (or flows) will be redirected to which slice, enabling for example, different applications to be routed through different slices. There exist many different approaches to provide slicing in SDN [Sherwood et al., 2009, Drutskoy et al., 2013, Bozakov and Papadimitriou, 2012, Kang et al., 2013, Yamanaka et al., 2014]. Among the studied languages, we identified four languages [Monsanto et al., 2013, Anderson et al., 2014, Soulé et al., 2014, Trois et al., 2015] providing this abstraction.

The *packet model* abstraction was found only in Pyretic [Monsanto et al., 2013]. In this language, each packet flowing through the network is a dictionary [Software Foundation, 2015] that maps field names to values. The dictionaries include entries for the packet location (either

physical or virtual), standard OpenFlow headers (e.g., source IP, destination IP, source port), and the virtual packet fields. Programmers can also add new virtual *fields* on packets header, and these fields are not limited to simple bit strings, they may also be arbitrary data structures. According to the authors, this extension provides a general way to associate high-level information with packets and to enable coordination between modules. In addition to extending the packet by including virtual fields, abstract packet model also extend packets by allowing every field (including non-virtual ones) to hold a stack of values instead of a single bit string. *Stacking* simulate a packet travelling through multiple levels of abstract networks. For example, to "lift" a packet onto a virtual switch, the run-time system *pushes* the location of the virtual switch onto the packet. Having done so, that virtual switch name sits on top of the concrete switch name. When the packet leaves the virtual switch, the run-time system *pops* a field off the appropriate stack.

Traffic Engineering

One can say that traffic engineering is concerned with performance optimization of operational networks. In general, it encompasses the application of technology and scientific principles to the measurement, modeling, characterization, and control of network traffic. Traffic engineering also includes the application of such knowledge and techniques to achieve specific performance objectives [Awduche et al., 1999]. Routing optimization plays a key role in traffic engineering, i.e., finding efficient routes to achieve the desired network performance [Wang et al., 2008]. QoS and resilience schemes were also considered as major components of traffic engineering [Akyildiz et al., 2014].

Kreutz et al. [Kreutz et al., 2015] identified many studies focusing on traffic engineering, most of them were classified as SDN applications. These works aim on different aspects of traffic engineering, such as QoS, load balancing, link failure, and energy saving. Akyildiz et al. classified the traffic engineering abstractions in a broaden scope: flow management, fault tolerance, topology update, and traffic analysis/characterization. We encountered languages providing abstractions to deal with *QoS*, *link failure*, and *path selection*.

The Quality of Service (*QoS*) can be defined as a set of service requirements to be met by the network while transporting a connection or flow [Ash, 2001]. These requirements can be expressed in terms of integrity constraints (packet loss), temporal constraints (delay), and timing restrictions for the delivery of consecutive packets belonging to the same traffic stream (delay variation) [Awduche et al., 2002]. In our survey, we perceived languages providing constructs to limit the bandwidth in terms of minimum and maximum usage. Minimum bandwidth allocates, to a specific traffic, a *guarantee* that the bandwidth will be more than the minimum specified value. Two languages implemented constructs to guarantee bandwith: FML [Hinrichs et al., 2009] and Merlin [Soulé et al., 2013a]. Maximum bandwidth is also named "bandwidth cap", and it allows the programmers to specify the maximum *limit* for certain flows. It was found only in Merlin.

One objective of traffic engineering is to simplify reliable network operations [Awduche et al., 1999]. Reliable network operations can be facilitated by providing mechanisms that enhance network integrity, and by embracing policies emphasizing network survivability [Awduche et al., 2002]. Network survivability refers to the capability of a network to maintain service continuity in the presence of faults. Some studies proposing fault-tolerance on SDN-based networks [Kim et al., 2012, Botelho et al., 2014, Chandrasekaran and Benson, 2014]. FatTire [Reitblatt et al., 2013] is the only studied language that provides abstractions to quickly change the forwarding behavior in cases of *link failure*. FatTire allows to specify sets of legal paths through the network, including the number of backup paths. However, it can handle only

link-level failures, not supporting device failures. FatTire compiler uses fast-failover groups to calculate every possible failure and precompute appropriate backup paths. Fast failover groups were introduced in OpenFlow v1.1+ to enable the switch to change forwarding without requiring a round trip time to the controller [ONF, 2014].

The last traffic engineering abstraction was called *path selection*. It enables programmers to define constraints on network paths, allowing to specify which paths the flows must or must not pass through the network. Path selection is being used in SDN networks to optimize load-balancing routing [Long et al., 2013] and QoS [Egilmez et al., 2013]. The abstraction was primarily implemented in FML [Hinrichs et al., 2009] by means of two language constructs: *waypoint* requires a flow to pass through a particular node in the network, and *avoid* that forbids a flow from passing through a particular node.

A regular expression is a specific text string used for describing a search pattern. It is often used for string matching, but it is also a natural and well-studied mathematical formalism for describing path selection through a graph [Fan et al., 2011]. Some network languages use *regular expression* to declare path constraints on sequences of network locations [Reitblatt et al., 2013] or packet transformations [Anderson et al., 2014, Soulé et al., 2014].

$$(tpDst = 22 \Rightarrow [*.IDS.*]) \\ \uplus (any \Rightarrow [GW.*.A])$$

This example is a FatTire [Reitblatt et al., 2013] program composed by two components. A security policy, given by the first line, which states that all SSH traffic must traverse the Intrusion Detection System (IDS); and a routing policy, given by the second line, which states that traffic from the gateway (GW) must be forwarded to the access switch (A), along any path.

Monitoring

One advantage of programming the network is the possibility to change its behavior due to some event, for example, changing the routing because a link becomes overloaded. To implement this, it is necessary to read information from the network. Some languages present a specific *query language* to gather the network data. Other languages permit to query the network considering a specific period of time (e.g., last 5 minutes). We used the term *windowed history* for these window-based abstractions. Besides the languages, there are publications related to monitoring the network information [Tootoonchian et al., 2010, Yu et al., 2014, Van Adrichem et al., 2014, Chowdhury et al., 2014].

Some languages present a specialized subset of constructs to query network information. This *query language* allows the programmers to express the statistics needed in their programs. Query language raises the level of abstraction and frees the programmers from worrying about the way the query rules are installed on the switches. It also allows the program to be splitted in querying information and policy specification. Query languages include constructs for filtering a set of packets in the network, grouping results by header fields, aggregating by number or size of packets, and limiting the number of values returned. This grouping is interesting, for instance, to instruct the switches to redirect certain flows to specific network functions such as Deep Packet Inspection (DPI). Some surveyed languages provide a sublanguage specifically to query the network [Foster et al., 2010, Katta et al., 2012, Foster et al., 2013, Nelson et al., 2014].

Another abstraction allows *windowed history* queries on the network. These windows can be defined in terms of time, number of packets, or packet size. Many programs need to receive periodic information about traffic statistics, for example, the programmer may use a windowed query to collect information minutely. If the language does not support this abstraction,

it would be necessary to check manually switch counters, and registering triggers to inquire the counters repeatedly every minute.

$Select(bytes)$
$GroupBy([srcip])$
$Every(60)$

This program, written in Frenetic [Foster et al., 2010], looks at all traffic, groups by source IP address, and aggregates the number of bytes every 60 seconds. This abstraction was found in eight programming languages [Foster et al., 2010, Voellmy et al., 2012, Monsanto et al., 2012, Foster et al., 2013, Reich et al., 2013, Anderson et al., 2014, Soulé et al., 2013b, Kim et al., 2015].

We named as *External Interface* the ability of the SDN languages to access information beyond the scope of the language. This information can include external references to SQL queries over databases, hash tables, and arbitrary code written in another language. It can also include external events originated, for example, by intrusion detection systems or authentication servers. We identified six languages that explicitly claim to provide support to access external information and events [Hinrichs et al., 2009, Voellmy et al., 2012, Monsanto et al., 2012, Nelson et al., 2014, Trois et al., 2015, Kim et al., 2015].

2.3.5 Northbound Related Work

This section presents some other works offering northbound abstractions. We selected some relevant efforts that, despite not having the language as a main aspect, propose frameworks or APIs to raise the level of abstraction in different areas. These works are succinctly described, ordered by their publication date.

Gutz et al. [Gutz et al., 2012] proposed network isolation at the language level. They created a library in Python that allows to instantiate virtual topologies through Mininet[15], and to define isolated slices on top of these topologies. Different SDN applications can be associated with different slices. PANE [Ferguson et al., 2012b, Ferguson et al., 2013] provides a Northbound API for delegating privileges for users to request network resources (e.g., bandwidth or access control). It uses a hierarchy conflict resolution supplied by HFT [Ferguson et al., 2012a].

Trema [Shimonishi et al., 2012] is a framework covering the entire development cycle of programming, testing, debugging, and deployment. FRESCO [Shin et al., 2013] is another framework, but focusing on developing and deploying security services. It provides some security functions, such as firewalls, scan detectors, attack deflectors, and IDS. Maple [Voellmy et al., 2013] is also a framework that, through an API, allows the programmer to use a standard programming language for programming OpenFlow switches. Maple optimizer observes the algorithm execution traces, organizes these traces to develop a partial decision tree. After that, it compiles these trees into optimized flow tables for distributed switches. Maple includes the McNettle OpenFlow network controller [Voellmy and Wang, 2012] that efficiently executes user-defined OpenFlow event handlers on multicore CPUs. Maple could be considered as an imperative programming language just like Pyretic, but the authors focused on other aspects, hindering us to classify it as a language.

NVP [Koponen et al., 2014] is a network virtualization platform to manage virtual networks in multi-tenant data centers. It uses a declarative language called nlog for computing the network forwarding state. NVP users cannot use this language, it was used only internally to develop NVP. Path-queries [Narayana et al., 2014] is a query language implemented on top of

[15]http://mininet.org/

Pyretic [Monsanto et al., 2013]. It uses a regular-expression-based path language that includes SQL-like *groupby* constructs for counting aggregation. It provides some useful applications, including single path packet count, traffic matrix generation, and congested link detection.

Concurrent NetCore [Schlesinger et al., 2014] was inspired in NetCore [Monsanto et al., 2012]. It is a programming language allowing to describes the layout of switches' flow tables. Currently, the switches do not have programmable tables, but P4 [Bosshart et al., 2014] proposed them to be configured according to the user's needs. The language acts mostly in CDPI layer, providing a small number of northbound operations for specifying packet processing, plus modular composition operators. The language introduces the concurrent compositor. While parallel compositor makes copies of packets, and then performs the actions on theses copies, the concurrent compositor allows two policies to act simultaneously on the same packet, so non-conflicting actions may be executed concurrently to reduce packet processing latency.

2.3.6 Technical Information and Summarization

In this section, we summarized the surveyed SDN languages, describing their technical information, evolutions, and relationships. Figure 2.18 shows the genealogy of these languages. Each node represents a programming language. The relationship between them was based on the information contained in their papers, websites, and source code repositories. The dates were also estimated considering these three information sources. A language has a continuity line if it was renamed or if it provides a source code repository; in this case we also plotted its last commit. If a language appeared only once (publication date), it was represented by just the node, without the continuity line.

One can perceive that there are two peculiar situations: (1) FlowLog [Nelson et al., 2013] was initially implemented on top of NetCore [Monsanto et al., 2012], but at a given time, Frenetic-OCaml [Foster et al., 2013] replaced NetCore. The approximate time of this modification is shown in genealogy with an arrow. The same happened with the (2) Frenetic-OCaml, which replaced the NetCore by NetKAT. We believe that these changes occurred due to the discontinuity of NetCore.

An overview of the languages development state is reported in Table 2.5. It includes the license of open-source languages, in which language they were implemented, their repository (Rep), the number of commits (Comm), branches (Brn), releases (Rel), contributors (Cont), and the last commit. This table along with Figure 2.18 and Tables 2.3 and 2.4 provide relevant information in choosing the best language to develop this proposal.

To better identify how languages evolved, we categorized them into two main groups as follows: (1) *specific purpose* languages, aiming to solve a particular problem, providing specific operations; and (2) *general purpose* languages that allow a more general and wider set of operations.

As *specific purpose*, we labeled the following languages: FatTire [Reitblatt et al., 2013], Merlin [Soulé et al., 2013a], PonderFlow [Batista and Fernandez, 2014], and NoF [Trois et al., 2015]. FatTire was created for writing fault-tolerant network programs. The language permits to specify allowable backup paths to be used in case of link failure. Merlin was designed to allow programmers to set the intended behavior of the network in terms of legal paths and bandwidth requirements (limiting and guaranteeing). The language provides constructs to support time-based windows and aggregation. It also includes mechanisms for delegating sub-policies and verifying if delegated sub-policies do not violate global constraints. PonderFlow was developed to specifically provide access control. Its constructs enable programmers to

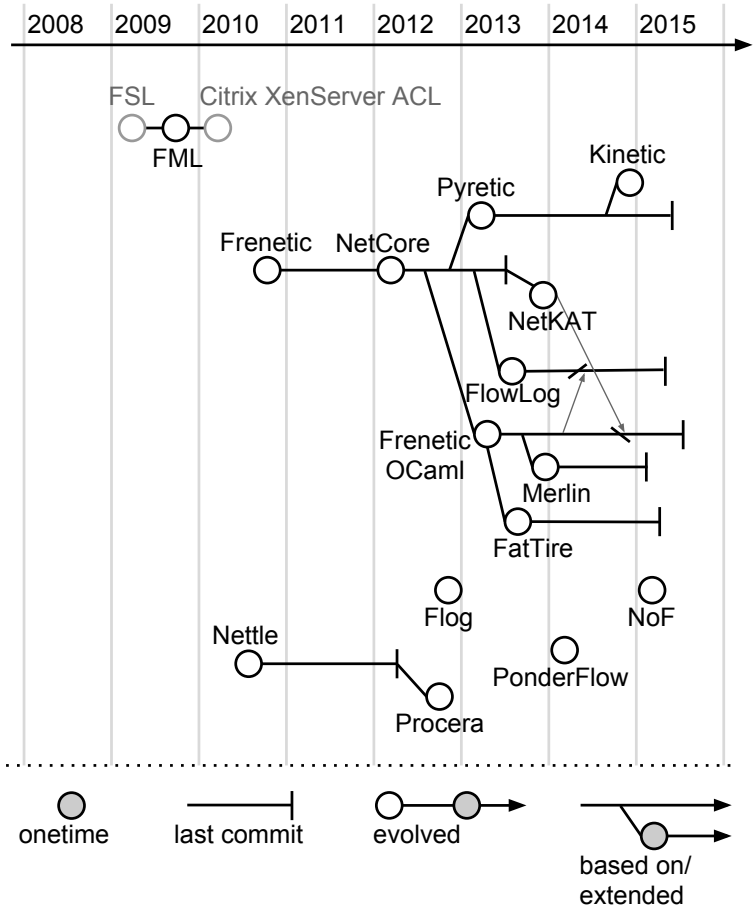


Figure 2.18: SDN languages genealogy.

define which users can access and modify switches information, as well as, add or remove rules in their flow tables. NoF provides abstractions to support the applications running on the top of the network. Its language provides constructs for reading and modifying host information, dealing with QoS priority, and redirecting flows through virtual topologies.

The *general purpose* languages were subdivided in three groups considering their programming paradigm: *event-driven*, *imperative* and *functional*. The *event-driven* languages use the concept of states. The authors of these languages stated that finite state machines are amenable to model checking, and also some Internet protocols [Lang, 2005, Rekhter et al., 2006] were implemented using state machines. Another important aspect is that the event-driven model is close to the lower level layer, and the translation to OpenFlow rules would be simpler. However, by using only finite state languages, as proposed by Flog [Katta et al., 2012] becomes impractical for two reasons: One reason is because these languages provide restricted expressive power. FlowLog [Nelson et al., 2014] dealt with this issue by combining elements from both restricted and full languages. It provides interfaces and abstractions for interacting with external programs. The second problem occurs as the size of the network increases; the large number of hosts, flows, network events, and policies may cause a state explosion. To deal with this situation, Kinetic [Kim et al., 2015] introduced an abstraction called Located Packet Equivalence Class (LPEC). The programmer creates a generic FSM, uses LPEC to specify a division of the flow space (e.g., all flows from the same source MAC address), and maps the LPEC to the generic FSM. Kinetic instantiates multiple copies of the generic FSM, one per LPEC. Kinetic also allows to compose FSMs by using the Pyretic's inherited modular compositors.

Table 2.5: SDN programming languages development information.

Language	License	Implemented In	Repository	Commits	Branches	Releases	Contributors	Last commit
FML [59]	—	C++, Python	—	—	—	—	—	—
Nettle [60]	SD-3-Clause	Haskell	https://github.com/AndreasVoellmy/nettle-openflow	30	1	0	1	Jul 27, 2015
Frenetic [61]	—	Python	—	—	—	—	—	—
Procera [62]	—	Haskell	—	—	—	—	—	—
Flog [63]	—	—	—	—	—	—	—	—
NetCore [69]	BSD-3-Clause	Haskell	https://github.com/frenetic-lang/netcore-1.0	598	4	1	9	Nov 3, 2014
Frenetic-Ocaml [70]	LGPL-3.0	OCaml	https://github.com/frenetic-lang/frenetic	4781	36	12	31	Dec 14, 2015
Pyretic [53]	BSD-3-Clause	Python	https://github.com/frenetic-lang/pyretic	1708	18	1	14	Aug 29, 2015
FlowLog *v2* [85]	LGPL-3.0	OCaml	https://github.com/tnelson/FlowLog	1025	1	1	3	Apr 10, 2015
FatTire [73]	LGPL-3.0	OCaml	https://github.com/frenetic-lang/fattire	754	1	0	5	Mar 21, 2015
NetKAT [71]	LGPL-3.0	OCaml	https://github.com/frenetic-lang/netkat	61	3	0	3	Dec 3, 2013
Merlin [57]	LGPL-3.0	OCaml	https://github.com/merlin-lang/merlin	5	1	0	1	Jan 21, 2015
PonderFlow [65]	—	Java	—	—	—	—	—	—
NoF [66]	—	Python	—	—	—	—	—	—
Kinetic [74]	BSD-3-Clause	Python	https://github.com/frenetic-lang/pyretic/tree/kinetic	759	6	1	14	Sep 26, 2014

The *imperative* programming paradigm was adopted by Pyretic language [Monsanto et al., 2013]. This language proposed network virtualization in terms of virtual switches and slicing. Pyretic also inovated being the only language to propose an abstract packet model with virtual fields and packet header stacking (Section 2.3.4). Although the imperative paradigm is more prevalent for programming computers, in the context of networks, it does not seem to be the most appropriate choice, mainly due to be far from networks programming model.

The remaining languages were defined as *general purpose* and *functional* (reactive), with one exception: FML [Hinrichs et al., 2009]. FML was the first SDN programming language; it included constructs for defining paths through the network, imposing bandwidth limits, and also allowing policies to be written by using external information sources. Besides FML was implemented using the logic paradigm, we fitted it with this group because it provides general purpose abstractions, and it inspired some other languages in this group. Nettle [Voellmy et al., 2010] introduced dynamic policies and modular composition. Frenetic [Foster et al., 2010] incorporated arbitrary wildcards, proposed a separate language to query network information, and also included windowed history queries. Procera [Voellmy et al., 2012] extended Nettle by adding an interface to deal with external events. NetCore [Monsanto et al., 2012] was at the forefront by presenting formal semantics and proofs of correctness. It was used as the core language to express forwarding in Pyretic and Frenetic-OCaml. NetCore also innovated by proposing proactive flow installation. Frenetic-OCaml [Foster et al., 2013] was the first language to provide constructs to ensure consistency [Reitblatt et al., 2011]. Finally, NetKAT [Smolka et al., 2015] incorporated virtual topologies, regular expression for describing legal paths through the network, and the iteration compositor. NetKAT replaced NetCore in Frenetic-OCaml.

2.4 Chapter Remarks

SDN is a paradigm that proposes the generalization of network devices, providing a standard set of functions for programming them. SDN decouples the network software from hardware, implementing a logic centralized controller that is responsible for programming the forwarding devices. In this chapter we outlined the SDN architecture, exposing its layers, components, interactions, and functionalities.

As seen in this chapter, OpenFlow is the most used SDN CDPI, but its interface is low-level and programming the network by using only the OpenFlow protocol may be error-prone. To combat this problem, controllers are raising the network programming level through APIs and programming languages. These different levels of abstraction were also described in this chapter, and examples were used for developing simple network applications, discussing the differences and advantages of each abstraction level.

This chapter also presented a comprehensive survey on SDN programming languages, revealing a broad range of features, but at the same time, there are a considerable amount of intersections among them. We perceived a lack of consensus on the terminology used in abstractions provided by these languages, which often makes extremely hard to compare and classify them. It motivated us to carefully investigate the language's features, proposing a taxonomy for grouping the similar ones. We identified nineteen different features, divided into ten categories. Our taxonomy led to a coherent view on the state-of-the-art of network programming languages, contributing to the advance of NBI standardization.

In the next chapter, we present efforts characterizing the communication patterns expressed by HPC applications. We also report an investigation of current SDN proposals for optimizing HPC applications, first describing the advantages and issues introduced by SDN and then, we associate this with related worke that somehow use SDN for optimizing the applications.

Chapter 3

Application Optimization and Communication Patterns

HPC applications are used in different domains, including bioinformatics, astrophysics, weather forecasting, genome research, among others. The performance of these applications is highly dependent on the interconnection bandwidth for the computing nodes and, as reported in Chapter 2, SDN emerged to support new possibilities for network management and configuration, enabling the network to become directly programmable according to the user requirements [Jammal et al., 2014].

This chapter describes the benefits that an SDN can bring for HPC applications, relating them to current relevant works that modify the network aimimg to improve the performance of these applications. This chapter also presents the efforts aiming to identify the applications' communication patterns, reporting studies that classify the applications according to them. Lastly, the computational techniques used for automatically recognizing the applications are also investigated.

3.1 SDN for Optimizing Applications

SDN allows the network to be modified for attending the demands of specific applications. For example, applications exchanging high volumes of data can benefit from an increase in the bandwidth if their traffic flows are balanced through the network topology paths. Multicast and broadcast messages are often used in synchronization mechanisms, involving small messages that are latency bound; so, speeding up these messages result in a better application performance.

On the one hand, SDN introduces many benefits such as flexibility in routing and centralized view of the network; on the other hand, by enabling the specification of matching using multiple packet header fields, SDN posed some new issues such as the time for populating the flow tables. Table 3.1 enumerates the advantages and issues introduced by SDN which may improve or impair the performance of HPC applications.

We studied some works proposing SDN applications which, in one way or another, presents an approach for optimizing the network according to the requirements of the application running on top of the network. These works were are summarized in Table 3.2, grouped by their purpose, and briefly described in the next sections. Table 3.2 also relates the SDN advantages used and issues tackled by these efforts.

Table 3.1: Advantages and issues introduced by SDN.

Advantage	**Description**
(*A1*) Forwarding flexibilization	The ability to change on-the-fly the forwarding allows traffic to be better placed, for example, bandwidth bound flows may be balanced across the multiple network paths and latency-sensitive flows can be routed through the paths with the shortest round-trip time (RTT).
(*A2*) Centralized monitoring	By allowing the entire network to be visualized from a centralized point, SDN enables a better balancing of arriving flows, allows the computation nodes being properly relocated according to their communication patterns, and permits the switches' flow tables to be rearranged for optimizing the forwarding.
(*A3*) High-level abstractions	As shown in Section 2.3.4, high-level abstractions can streamline network management tasks such as security, consistency, and forwarding handling.
(*A4*) Dynamic QoS	The open and standard interface offered by SDN allows traffic to be dynamically shaping, so that, QoS policies may be enforced effortlessly for specific applications.
Issue	**Description**
(*I1*) Latency for populating the flow tables	The controller manages the switches' flow tables by adding, modifying, or removing their entries. The time for doing these operations can significantly affect network performance, especially if there are several short-lived flows.
(*I2*) Table lookup time	SDN-enabled switch would require 5 to 7 times more space on memory comparing to a typical layer-2 switching device [Kannan and Banerjee, 2013]. To alleviate this problem, SDN includes wildcard matching, using Ternary Content Addressable Memories (TCAM) [Yu et al., 2004] to speedup this process. This kind of memory is very expensive and power hungry; thus, the SDN vendors are using a combination of TCAM and software-based tables; the problem is that the latter is much slower than the TCAMs.

3.1.1 Adjusting the Network for Big Data Applications

Wang et al. designed an integrated a network control plane [Wang et al., 2012a] providing topology construction and routing mechanisms for some Hadoop operations including single aggregation, data shuffling, and partially overlapping aggregation. Narayan et al. proposed a modified version of the Hadoop scheduler (Hadoop-OFE) [Narayan et al., 2012] that adjusts QoS queues automatically before starting new tasks.

FlowComb [Das et al., 2013] is a framework to predict Big Data application network transfers. It uses software agents installed on application servers to collect information transparently to the application. A centralized decision engine collects data movement information

Table 3.2: SDN applications.

	Name	Target	Description	A1	A2	A3	A4	I1	I2
Big Data	Wang et al. [Wang et al., 2012a]	Hadoop	Topology construction and routing mechanisms for some Hadoop patterns	✓	✓				
	Hadoop-OFE [Narayan et al., 2012]	Hadoop	Presents a modified version of the Hadoop scheduler that automatically adjusts QoS queues before starting a task				✓		
	FlowComb [Das et al., 2013]	Hadoop	Predicts Hadoop transfers and schedules their flows to avoid link congestion	✓	✓				
	BASS [Qin et al., 2014]	Hadoop	Provides a bandwidth-aware task scheduler for Hadoop using time slots to guarantee bandwidh allocation		✓				
	Pythia [Veiga Neves et al., 2014]	Hadoop	Predicts Hadoop communications and uses a first-fit bin-packing heuristic to allocate the flows to available paths	✓	✓				
	Renner et al. [Renner et al., 2015]	Flink	Proposes a container placement approach that takes the network topology into account to prevent network congestions		✓				
	Firebird [He and Shenoy, 2016]	Spark	Proposes a network-aware scheduling method for Spark for avoiding data contention		✓				
Data Center	Hedera [Al-Fares et al., 2010]	Elephant flows	Detects large flows and uses placement algorithms (Global First Fit and Simulated Annealing) to compute paths	✓	✓				
	Wang et al. [Wang et al., 2011]	Load-balancing	Uses a partitioning algorithm to determines a minimal set of wildcard rules	✓	✓			✓	✓
	Mahout [Curtis et al., 2011]	Elephant flows	Detects elephant flows at the end-hosts by observing their socket buffers		✓				
	Vassoler et al. [Vassoler et al., 2012]	Changing topology	Modifies the physical topology to reduce the impact of multi-hop traffic forwarding	✓	✓			✓	
	MiceTrap [Trestian et al., 2013]	Mice flows	Provides flow aggregation and load balancing for mice flows		✓			✓	✓
	VMCluster [Kakadia et al., 2013]	VM colocation	Proposes an algorithm for clustering and placing VMs based on their traffic patterns		✓				
	B4 [Jain et al., 2013]	Multiple data center	Allows to define application priority and uses multipath routing/tunneling to optimize link usage	✓	✓	✓	✓		
	CloudMirror [Lee et al., 2014]	VM placement	Provides bandwidth guarantees by placing the VMs based on their communication structure		✓		✓		
	BwE [Kumar et al., 2015]	Bandwidth allocation	Provides bandwidth allocation with user-defined delegation, deal with failures and multipath forwarding	✓	✓	✓		✓	
	DLBS [Tang et al., 2017]	Load-balancing	Proposes a dynamical load-balanced scheduling (DLBS) for improving throughput and balancing network workload	✓	✓				
Application Aware	Jarschel et al. [Jarschel et al., 2013]	YouTube	Optimizes link utilization changing the path selection method	✓	✓				
	Das et al. [Das et al., 2011]	General applications	Provides packet flow agreggation for the same type of application and forwards them through specific paths	✓	✓				✓
	Atlas [Qazi et al., 2013]	General applications	Uses Machine Learning to identify the applications using the network		✓				
	OpenQoS [Egilmez et al., 2012]	Multimidia applications	Groups the incoming traffic as data and multimedia flows and forwards the multimedia through less congested links	✓	✓				✓
	Nam et al. [Nam et al., 2014]	Video streaming	Dynamically changes routing paths using bandwidth, packet loss rate and jitter to discovery congested links	✓	✓				
	NCP [Cofano et al., 2014]	Video streaming	Provides reserved bandwidth slices for single or aggregated video flows	✓	✓				
	Liu et al. [Liu et al., 2015]	General applications	Maintains a fixed database mapping of valid paths to each user application to speed up installation process	✓	✓			✓	
HPC	SDN-enhanced MPI [Takahashi et al., 2015]	MPI	Optimizes the network for some MPI primitives	✓	✓				
	Date et al. [Date et al., 2015]	MPI	Incorporates a network-aware job placement into SDN-enhanced MPI and uses SDN for rerouting on network failures	✓	✓				
	Polezhaev et al. [Polezhaev et al., 2014]	HPC	Proposes network-aware job placement, modifying the forwarding to improve the inter job communication	✓	✓				
	ASETS [Jamalian and Rajaei, 2015]	HPC	Proposes a task scheduling system for data-intensive HPC tasks		✓				
	SDN-HPC [Alsmadi et al., 2016]	HPC	Uses network slicing and proposes two load balance schemes for executing HPC jobs in parallel		✓				
	Wu et al. [Wu et al., 2016]	HPC	Reroutes flows on fat-trees avoiding links to be congested	✓	✓				

from agents and schedules upcoming flows on paths, avoiding the network does not become congested.

BASS [Qin et al., 2014] is a bandwidth-aware task scheduler to combine Hadoop with SDN. It first utilizes SDN to manage the bandwidth allocation in a time slots. The scheduler decides whether to assign a task locally or remotely depending on the completion time. Time slot bandwidth allocation guarantees that the bandwidth of all links on a path, on a given time slot, is reserved for a specific Hadoop task.

Pythia [Veiga Neves et al., 2014] can transparently predict Hadoop data communication volume at runtime. It uses this predictive knowledge to optimize the data center multipath network. Prediction intelligence is collected at a central controller and in turn ingested by a chain of network control algorithms (routing, flow scheduling) that optimize network resource allocation. Its approach uses the intermediate map file to estimate the amount of data will be used in the shuffle phase. It uses a first-fit bin-packing heuristic to jointly allocate sets of predicted shuffle transfer flows to available paths.

Renner et al. present a network-aware container placement approach for data analytics applications [Renner et al., 2015]. According to the authors, the key advantage of improving container placement is that workloads consisting of different frameworks, applications, and datasets in a shared cluster can benefit from this optimization. They use simmulated annealing to find acceptably good container placements based on a weighted cost function in a fixed amount of time.

He et al. proposed a network-aware scheduling method called Firebird [He and Shenoy, 2016], which optimizes Spark task scheduling based on network status of a cluster. Their scheduler allocates tasks to the nodes where the available bandwidth is equal or greater than that required by the task.

3.1.2 Data Center Network Improvements

Hedera [Al-Fares et al., 2010] is a dynamic flow scheduling system for multipath topologies. It detects large flows at edge switches, and then it estimates the natural demand of large flows and uses placement algorithms (Global First Fit and Simulated Annealing) to compute good paths for them.

Wang et al. [Wang et al., 2011] proposed a load-balancing architecture that proactively maps blocks of source IP addresses to replica servers. Client requests are directly forwarded through the load balancer with minimal intervention by the controller. They used a partitioning algorithm to determines a minimal set of wildcard rules to install, being also possible to configure balancing weights.

Mahout [Curtis et al., 2011] allows to detect elephant flows at the end-hosts, by observing their socket buffers. Once an elephant flow is detected, an end host signals the network controller using in-band signaling with low overheads. According to the authors, the combination of end host elephant detection and in-band signaling eliminates the need for per-flow monitoring in the switches, and hence incurs low overhead and requires few switch resources.

Vassoler et al. [Vassoler et al., 2012] proposed a reconfigurable physical layer with no link loss to provide higher throughput, and simultaneously, reduce CPU packet forwarding load in server-centric datacenters. They used inexpensive 2x2 and fast magneto-optical switches to transport heavy traffic and reduce the impact of multi-hop traffic forwarding by modifying the physical topology.

MiceTrap [Trestian et al., 2013] employs scalability against the number of mice flows through flow aggregation, together with a software-configurable weighted routing algorithm that offers improved load balancing for mice flows.

In large virtual data centers, the performance of applications is highly dependent on the communication bandwidth available among VMs. VMCluster [Kakadia et al., 2013] is an algorithm to cluster the VMs, based on their traffic exchange patterns. It also proposes an algorithm for placing the VM clusters such that, the application performance increases and the internal data center traffic is localized as much as possible.

B4 [Jain et al., 2013] is an application that enables to connect multiple data centers across the world. It leverages control at the network edge to adjudicate among competing demands during resource constraint. B4 uses multipath forwarding/tunneling to enhance available network capacity according to application priority, and dynamically reallocates bandwidth in the face of link/switch failures or shifting application demands.

CloudMirror [Lee et al., 2014] presents an abstraction based on application communication structure and not on the underlying physical network topology. This abstraction allow to specify the network requirements easily, it also facilitate the translation of these requirements to the low level infrastructure components. For guaranteeing bandwidth, CloudMirror incorporates a VM placement algorithm that meets bandwidth requirements specified by applications.

Bandwidth Enforcer (BwE) [Kumar et al., 2015] is a hierarchical bandwidth allocation infrastructure. BwE supports (1) service-level bandwidth allocation following prioritized bandwidth functions where a service can represent an arbitrary collection of flows. (2) Independent allocation and delegation policies according to user-defined hierarchy, all accounting for a global view of bandwidth and failure conditions. (3) Multipath forwarding common in traffic engineered networks, and (4) a central administrative point to override (perhaps faulty) policy during exceptional conditions.

DLBS [Tang et al., 2017] proposes a dynamical load-balanced scheduling (DLBS) approach to maximize the network throughput through dynamically balancing data flows. It uses different scheduling algorithms that quantitatively analyze the imbalance degree of data center networks at the beginning of each time slot and then schedule unbalanced data flows once a load imbalance happens.

3.1.3 Application Awareness

Jarschel et al. [Jarschel et al., 2013] optimized link utilization for YouTube videos by changing the path selection method. They studied Round-Robin, Bandwidth-Based, Deep Packet Inspection (DPI), and Application-Aware path selection approaches. In DPI proposal, the first packets are redirected to a DPI service, configured with regular expressions to identify YouTube flows. If a particular flow is a YouTube video, the controller will redirect the flow to another less congested link. Application-Aware path selection uses the same idea of DPI, but instead of regular expressions, it uses a tool for monitoring the quality of YouTube videos.

Das et al. [Das et al., 2011] provided differential treatment and dynamically aggregated packet flows for voice, video, and web traffic. They created virtual circuits through paths having characteristics beneficial for specific applications. For instance, VoIP traffic can benefit from low latency paths. For such bundles, they dynamically created a circuit between source-and-destination packet switches, where the circuit path is the one with the smallest propagation delay.

Atlas [Qazi et al., 2013] uses ML to identify applications using the network. It uses end-host agents to collect information about active network sockets, which are then sent to the controller. The controller applies an ML algorithm to identify the application by inspecting a specific set of flow features, such as the sizes of the first N packets, source and destination ports and IP addresses.

OpenQoS [Egilmez et al., 2012] enables QoS for multimedia delivery over OpenFlow networks. In order to support QoS, it groups the incoming traffic as data flows and multimedia flows, where the multimedia flows are dynamically placed on QoS guaranteed routes and the data flows remain on their traditional shortest path. OpenQoS does not use traditional QoS mechanisms; instead, it collects the network state information such as link speed, available bandwidth, and packet drop counts from switches, and it applies a formula for estimating congested links, routing the multimedia flows through less congested ones.

Nam et al. [Nam et al., 2014] proposed an application to monitor network conditions of streaming flow in real-time and dynamically change routing paths using multi-protocol label switching traffic engineering (MPLS-TE) to provide reliable video watching experience. It runs shortest path algorithm after selecting links that meet a given set of constraints (TCP bandwidth, packet loss rate, and jitter).

Network Control Plane (NCP) [Cofano et al., 2014] is another proposal to improve video streaming to jointly maximize users Quality of Experience and network utilization by allocating bandwidth on a per-flow basis. It uses Network Bandwidth Reservation (NBR) strategy to reserve a bandwidth slice to video flows.

Liu et al. [Liu et al., 2015] proposed a mapping scheme to route applications through the network. It maintains a fixed database mapping valid paths with user applications. When a new application flow is detected, their solution queries the database and installs the necessary flows, speeding up the installation process.

3.1.4 High-Performance Computing

The Message Passing Interface (MPI) library is a de facto standard for developing parallel and distributed programs [Achour and Nasri, 2012]. Takahashi et al. [Takahashi et al., 2015] implemented some SDN enhanced MPI primitives; they developed an MPI *Bcast* (broadcast) for eliminating duplicate packets in the network and an MPI *Allreduce* that makes a communication plan to forward the reducing through distinct paths. Date et al. [Date et al., 2015] discussed the integration of SDN with HPC infrastructure, extending Takahashi et al. [Takahashi et al., 2015] work by incorporating a network-aware HPC job placement. They also proposed using SDN for modifying the forwarding on network failures.

Polezhaev et al. [Polezhaev et al., 2014] proposed two modifications on BackFill job placement algorithm [Feitelson and Weil, 1998] for making it network-aware. They also used SDN for modifying the network to improve the inter job communication. Similarly, Jamalian et al. [Jamalian and Rajaei, 2015] proposed A SDN Empowered Task Scheduling System (ASETS) that allows to schedule data-intensive HPC tasks in a Cloud environment. They used the "bandwidth awareness" capability of SDN to better use the bandwidths when assigning tasks to virtual machines.

Alsmadi et al. [Alsmadi et al., 2016] proposed SDN-HPC, a model that allows executing multiple HPC jobs simultaneously by creating multiple network slices. Their approach observes the hosts that are exchanging high volumes of traffic, and avoids placing new HPC jobs on these hosts. Finally, Wu et al. [Wu et al., 2016] implemented a network congestion detector based on the traffic monitoring mechanism using SDN. They also designed a traffic flow scheduler that is capable of dynamically redistributing the network traffics for fat-trees rerouting congested path to the congestion-free ones.

3.2 Communication Patterns

Computing systems deployed for HPC applications have to process huge information datasets, and the communication is a significant performance factor. Kim and Lilja stated that a proper understanding of the communication patterns in this kind of applications is essential for determining how to maximize their effectiveness within a given environment [Kim and Lilja, 1998]. So, we revisited the publications studying the communication patterns.

In 1993, Cypher et al. studied the behavior of parallel applications, quantifying the number of messages and volume transmitted in several applications [Cypher et al., 1993]. They found three communication patterns binary tree, hypercube, and torus, concluding that the communication patterns of those applications could be known at compilation time and remained fixed throughout in all ran.

In 1997, Chodnekar et al. developed a framework for characterizing the communication properties of parallel applications, based on message frequency, spatial distribution and length [Chodnekar et al., 1997]. Kim and Lilja quantified the communication patterns, characterizing the traffic by using message destination, size, frequency distributions and message locality [Kim and Lilja, 1998]. The locality was defined as a repetition of communication events (send or receive), message size and message destination.

Vetter and Mueller examined the communication characteristics of several applications, from the perspective of Message Passing Interface (MPI) [Vetter and Mueller, 2003]. They characterized the applications in point-to-point and collective communication. In point-to-point communication, they measured the number of messages, type, payload size, and destination. For collective communication, they determined the type, frequency, and payload size.

Beson et al. conducted an empirical study of the network traffic in ten data centers, including university, enterprise, and cloud data centers [Benson et al., 2010]. They collected and analyzed Simple Network Management Protocol (SNMP) statistics, topology, and packet-level traces. They examined the applications deployed in these data centers and their placement, reporting the flow-level and packet-level transmission properties and their impact on network utilization, link utilization, congestion, and packet drops. They concluded two important implications for the SDN centralized approach. First, due to the fact of a small number of active flows, OpenFlow-switches can maintain few rules in their flow tables. Second, high frequency of flow inter-arrival times has important implications for the scalability of the controller, suggesting the use of multiple controllers or proactive flow installation (Section 2.3.1).

Colella was beyond of just identifying traffic patterns using some package header information. He identified seven important numerical methods for science and engineering. Satirizing Snow White and the Seven Dwarfs, he named them as seven dwarfs [Colella, 2004]. He formerly called them as Dense Linear Algebra, Sparse Linear Algebra, Fast Fourier Transform, Structured Grids, Unstructured Grids, Particles, and Monte Carlo. These methods are also known as computational motifs and they constitute classes where membership in a class is defined by similarity in computation and data movement.

Asanovic et al. studied several different applications (such as benchmarks, database, games and ML) and updated Colella's list [Asanovic et al., 2006]. They renamed three dwarfs and added six more dwarfs. Figure 3.1 shows the original seven dwarfs plus the new ones[1]. Fast Fourier Transform was renamed to Spectral Methods, Particles was renamed to N-Body Methods, and Monte Carlo was called MapReduce. These new dwarfs were called Combinational Logic, Graph Traversal, Dynamic Programming, Backtrack & Branch+Bound, Construct Graphical

[1]Adapted from: `http://developer.amd.com/wordpress/media/2013/06/2155_final.pdf`.

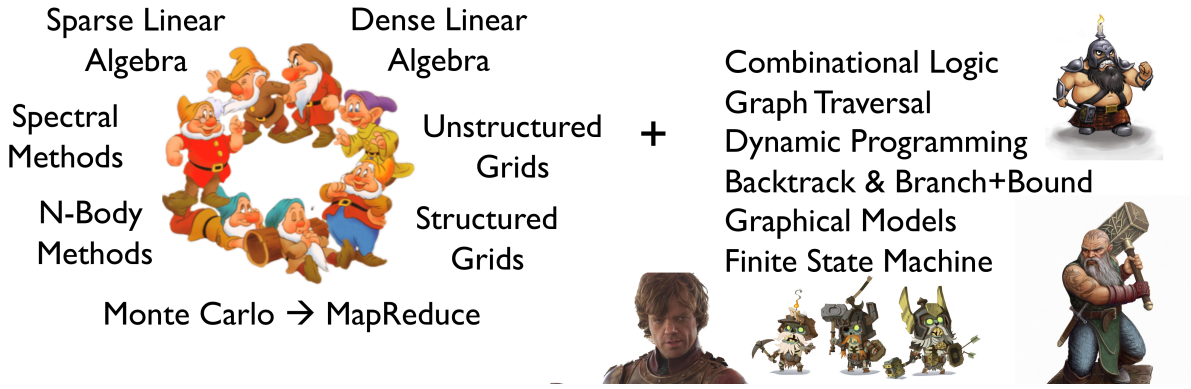


Figure 3.1: The seven dwarfs of Colella and the six dwarfs of Berkeley.

Models, and Finite State Machine. Distributed parallel programs fall into one or more of these 13 dwarf classes and, if the variance of the expressed patterns is bounded, identification of the dwarf class should be possible solely from observed communications [Whalen et al., 2012].

The dwarfs are widely used in many computational areas such as for designing network-on-chip [Stuart et al., 2011], encompassing coherence protocols [Barrow-Williams et al., 2009], and thread mapping [Diener et al., 2015], and aiming to speed up user applications [Barkai, 2009, Beach, 2010, Solano-Quinde et al., 2011, Rubin et al., 2014].

Importance of Dwarfs on Applications Areas

There exist several works aiming to classify the applications according to dwarfs computation methods [Asanovic et al., 2006, Ahern et al., 2007, Che et al., 2009, Springer, 2011, Feng et al., 2012, Whalen et al., 2012]. We studied these works, summarizing the use of dwarfs in computational areas (Figure 3.2).

Dwarf	Application Areas
1. Dense Linear Algebra	Linear Algebra
	Data Mining
2. Sparse Linear Algebra	Finite Element Analysis
	Partial Diferential Equations
3. Spectral Methods	Fluid Dynamics
	Quantum Mechanics
	Weather Prediction
4. N-Body Methods	Molecular Modeling
	Molecular Dynamics
	Cosmology
5. Structured Grids	Image Processing
	Physics Simulations
6. Unstructured Grids	Fluid Dynamics
	Belief Propagation
7. MapReduce	Distributed Searching
	Sequence Alignment
	Parallel Monte Carlo Simulations

Dwarf	Application Areas
8. Combinational Logic	Encryptation & Decryptation
	Hashing
9. Graph Traversal	Searching
	Sorting
	Collision Detection
10. Dynamic Programming	Graph Problems
	Sequence Alignment
11. Backtrack &B+B	Artificial Intelligence
	Integer Linear Programming
	Boolean Satisfiability
	Combinatorial Optimization
12. Graphical Models	Sequence Homology Search
	Machine Learning
	Embedded Computing
13. Finite State Machines	Video Decoding, Parsing, Compression
	Data Mining
	Find Repeating Patterns

Figure 3.2: Use of dwarfs on application areas.

Asanovic et al. applied the dwarfs to a large number of computational applications [Asanovic et al., 2008]. They compared the dwarf classes against collections of benchmarks

for embedded computing (42 benchmarks EEMBC[2]) and for desktop and server computing (28 benchmarks SPEC2006[3]). They also examined relevant application domains: artificial intelligence/ML, database software and computer graphics/games. Figure 3.3[4] shows the temperature chart of need of these applications. We can see that HPC applications make heavy use of the original dwarfs proposed by Colella, presumably because this is the area of expertise of this researcher.

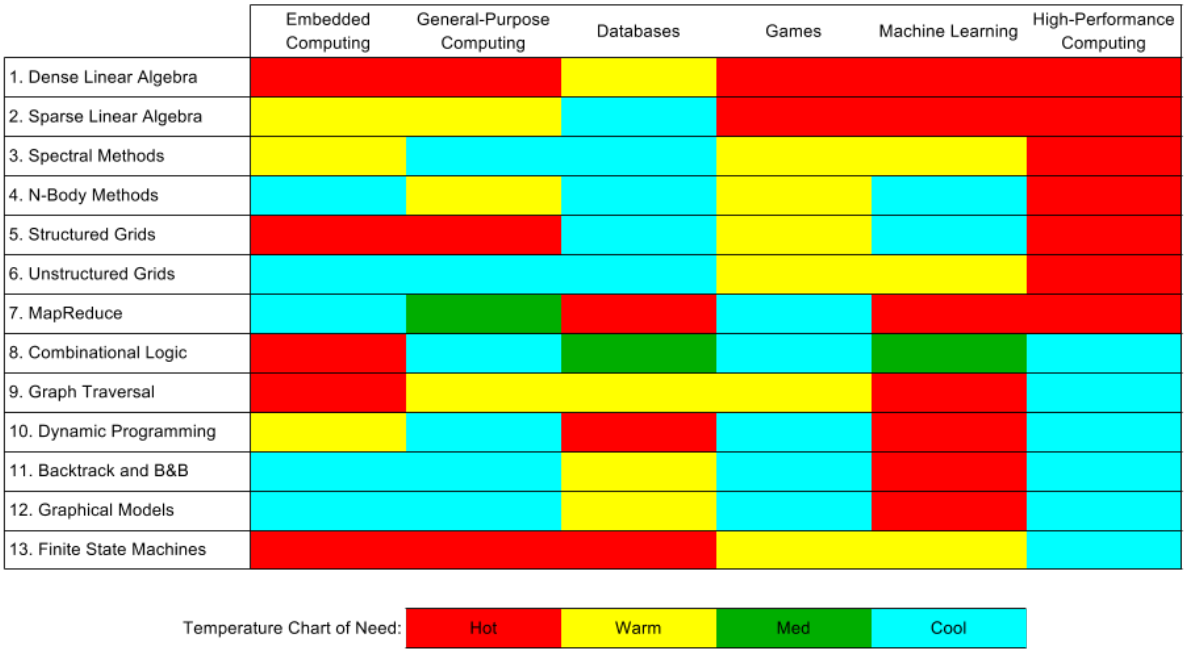


Figure 3.3: Temperature chart of dwarf usage in applications.

Another relevant aspect to be considered is that the applications may have multiple execution phases; namely, their computational workloads and communication patterns may change at runtime [Shan et al., 2000]. For example, a MapReduce task divide in two phases where the first phase maps a user supplied function to thousands of computers, processing key/value pairs to generate a set of intermediate key/value pairs. The second phase reduces the returned values from all those thousands of instances into a single result by merging all intermediate values associated with the same intermediate key.

Communication Requirements

The applications' communication requirements were investigated by Asanovic et al. [Asanovic et al., 2006]. They observed that most point-to-point communications were stable and sparse, and they are typically large enough to remain strongly bandwidth bound. Therefore, this type of message would prefer a dedicated point-to-point pathway through the interconnect to minimize the chance of contention within the network fabric. Another caveat is that the point-to-point messaging tend to utilize only a fraction of the available communication paths through a fully connected network switch fabric.

Collective communication requirements are sharply differentiated from point-to-point, as they tend to involve very small messages that are primarily latency bound. This propo-

[2]https://www.eembc.org/
[3]https://www.spec.org/cpu2006/
[4]Adapted from: Asanovic et al. [Asanovic et al., 2008]

sition was also detected by Benson et al. in their empirical study of network traffic in data centers [Benson et al., 2010].

Although the communication patterns are sparse, they are not necessarily isomorphic. Asanovic et al. concluded that "assigning a dedicated path to each point-to-point message transfer is not solved trivially by any given fixed-degree interconnect topology" [Asanovic et al., 2006]. To this end, they proposed two solutions: (1) carefully place jobs so that they match the static topology of the interconnect fabric, and (2) employ an interconnect fabric that can be reconfigured according to the application's communication topology.

3.2.1 Spatial and Temporal Behaviors Through Traffic Matrices

An interesting approach to be considered when analysing the communication patterns is the measurement of the TM. A TM represents the amount of traffic between origin and destination in a network. It has a tremendous potential utility for many IP network engineering applications, such as network survivability analysis, traffic engineering, and capacity planning [Zhang and Ge, 2005].

We investigated the communication patterns of a set of benchmarks developed by NASA Advanced Supercomputing Division, called NAS Parallel Benchmarks (NPB)[5] [Bailey et al., 1991], selecting those that most exchanged information among the computation nodes: bt, cg, ft, and lu[6], and analyzing their spatial and temporal behaviors.

Spatial Behavior

Spatial behavior can be formalized as a traffic matrix M_B, where each position $M_B[i][j]$ holds the number of bytes transmitted from node i to node j during a specified period of time. For the graphical visualization, we normalized the matrix to its maximum value, showing it in a gray gradient, where cells in black are the most communicating pair of nodes and white means that no communication happened. All traffic matrices shown in this thesis are displayed using this grayscale.

Figure 3.4 gives the spatial behavior of selected applications, showing the total amount of data exchanged throughout their execution time. It is possible to see that these applications feature different spatial communication behavior; *ft* exchanges almost the same amount of data across all nodes, transmitting a maximum of 185.7 MB by a pair of nodes. *cg* is the program that most exchanged data, considering the pairs of nodes (597.1MB), however, as we can see on its spatial behavior matrix, only a few pairs of nodes communicated. The total amount of data transmitted from all computation nodes, during the entire application execution, is also displayed.

---

[5]The NAS Parallel Benchmarks are set of programs designed to help evaluate the performance of parallel supercomputers, consisting of five kernels and three pseudo-applications. Available at: `https://www.nas.nasa.gov/publications/npb.html`

[6]The *bt* algorithm is used to compute block tridiagonal matrices; it is often used to solve engineering problems, and it is classified as Dense Linear Algebra. The *cg* method is used for computing an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix; it is an Unstructured Grid dwarf. *ft* solves a 3-D partial differential equation using Fast Fourier Transform; it is classified as Spectral Methods. *lu* uses iterative methods for solving linear systems; it was classified as Sparse Linear Algebra.

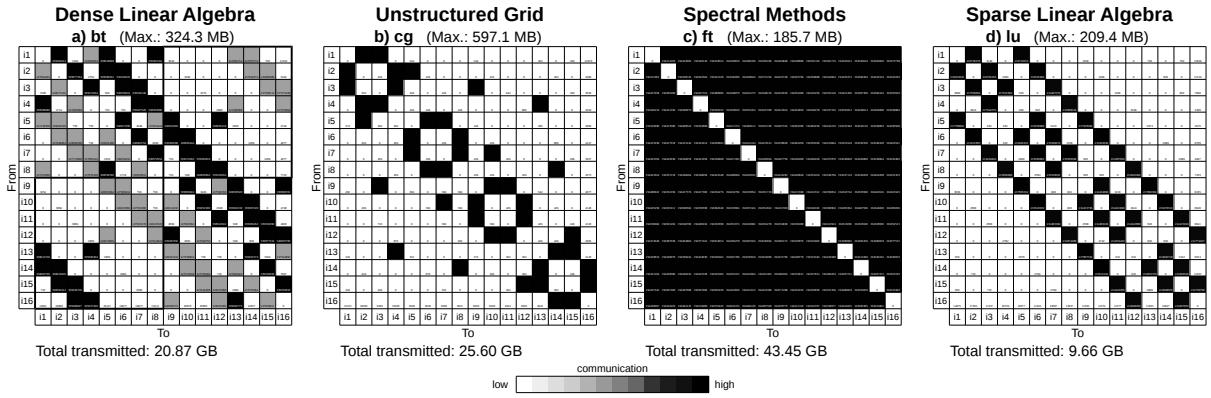


Figure 3.4: Spatial behavior through traffic matrices.

Temporal Behavior

Includes the notion of time to M_B. It can be formalized as $M_T[t](M_B[i][j])$, where each instant t holds a spatial behavior matrix $(M_B[i][j])_{t''}$, filled with the amount of data transmitted during the interval $[t'' - t']$, where t' is the instant of time just before t''.

Figure 3.5 shows a graphical representation of M_T for the *ft*. At the moment $t(1)$, it is initializing, and there is an intensive communication among the master node $i8$ with all other nodes. In the second moment ($t(2)$), the node $i1$ starts transmitting to $i2$, and the following moments, it transmits to $i3$, $i4$, and $i5$, successively.

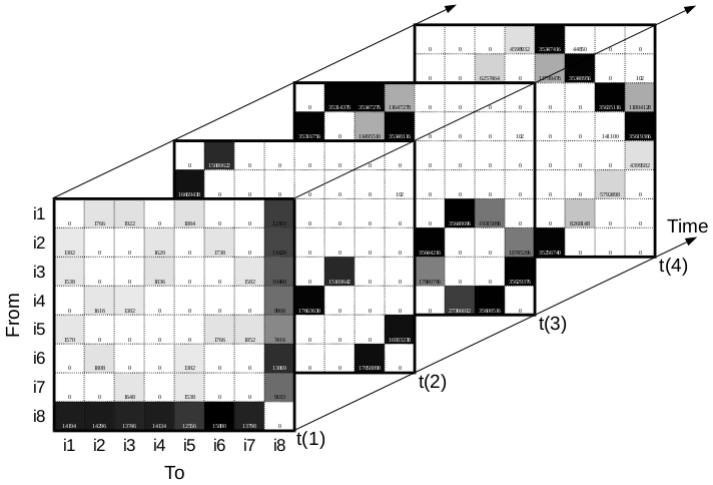


Figure 3.5: 3-D temporal behavior representation.

X-axis slicing the M_T allows seeing how a specific node transmits data over time. Figure 3.6 shows the x-axis slice of *cg* and *ft*. As can be seen, the selected node of *cg* ($i7$) is transmitting data to a few nodes ($i5$, $i8$, and $i10$) at an almost constant rate. On the other hand, *ft* has a different behavior and the selected node ($i1$) is transmitting, in sequence, to all the other nodes.

Discussion

We executed these programs multiple times, varying the data input size and the number of computing nodes (Figure 3.7). We perceived that, regardless of the input data or the number

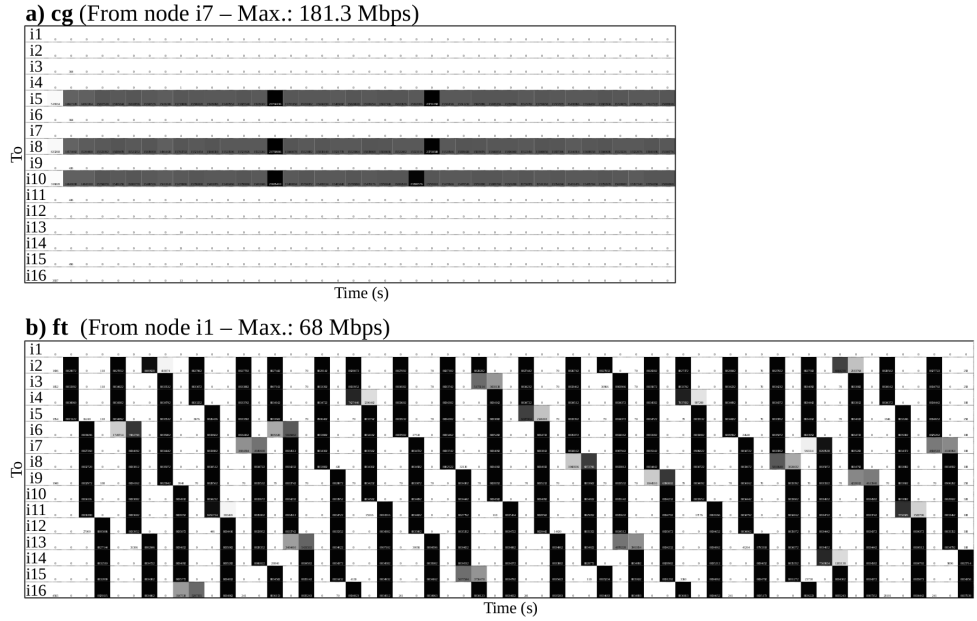


Figure 3.6: Temporal behavior of specific transmitting nodes.

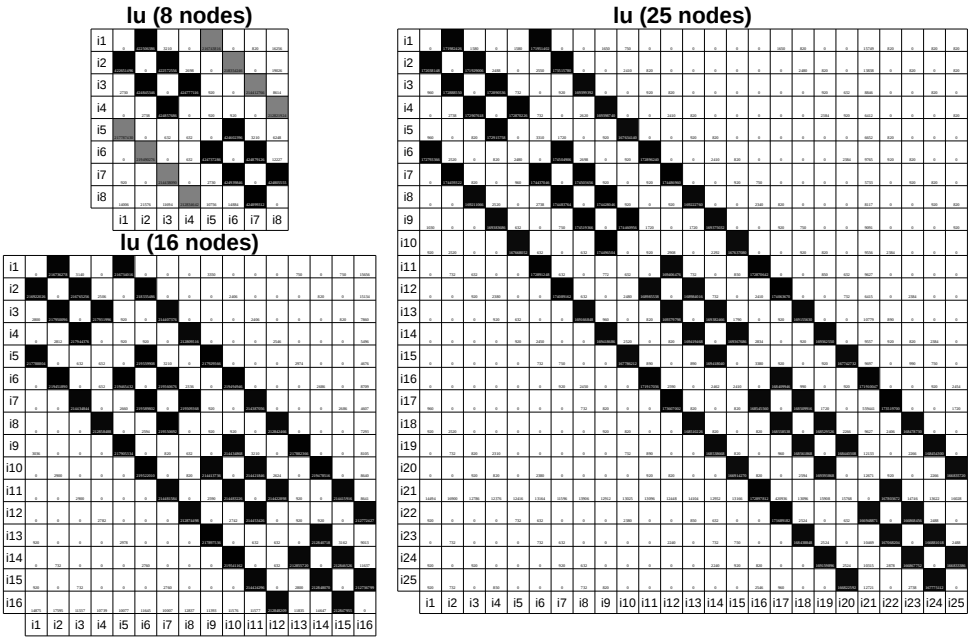


Figure 3.7: Communication behavior varying the amount of computing nodes.

of computing nodes, it was possible to visually identify different patterns on their TMs. Except for the MapReduce dwarf, all other tested HPC applications and benchmarks presented a "well-behaved" communication pattern, meaning that a given HPC application executed in a set of nodes has a strong trend to transmit the same amount of data across the same nodes.

We also ran several applications classified as the same computational dwarf, realizing that each application maintained a specific communication pattern, but we could not observe similarities among the different applications' TMs.

3.2.2 Computational Techniques for Classifying the Communications

As stated earlier, HPC applications exchange an enormous volume of data, quickly stressing the capabilities of networks, so automated methods for understanding of the communication demands and classifying these applications became a major challenge in networking research [Srivastava et al., 2016].

Existing proposals for detecting the communications fall into three categories where each one has its own Achilles' heel. The first approach uses port-based classification process and has been recognized as being inaccurate as many applications have adopted dynamic port numbering to overcome the performance limitations of networks [Dhote et al., 2015].

Another approach is based on DPI which consists of examining the payloads of the packets for classifying traffic. This approach not only imposes significantly higher computational complexity but also requires specific knowledge of the application protocols [Bujlow et al., 2013]. Furthermore, many applications, such as HPC applications, may adopt cryptographic methods for ensuring security in the communication among their computing nodes and therefore preventing DPI from classifying any traffic.

In a third category, ML techniques have been used for traffic classification; these methods work by exploiting intrinsic and statistical flow information as a representation for feeding the ML classifiers. For instance, packet size average and variance, total number of packets or bytes, flow duration, server and client port numbers. So, we selected some relevant works which have achieved significantly high classification accuracy.

Bernaille et al. proposed a technique using an unsupervised ML (Simple K-Means) algorithm that classified different types of TCP-based applications using the first few packets of the traffic flow [Bernaille et al., 2006]. Their tests aimed to classify ten applications, and their approach was able to identify more than 80% of the applications flows correctly. Eerman et al. classified the traffic using supervised (Naïve Bayes classifier) and unsupervised learning (Expectation Maximization clustering algorithm) [Eerman et al., 2006]. Their testbed was composed of traces collected from University of Auckland which tried to classify eight applications. Their results show an accuracy of 91% when using unsupervised learning and 82% with supervised learning.

Soysal and Schmidt investigated and evaluated the classification performance of three supervised ML algorithms (Bayesian Networks, Decision Trees, and Multilayer Perceptrons) for classifying six different types of traffic [Soysal and Schmidt, 2010]. Their datasets were acquired from the National Academic Network of Turkey, and the accuracy reported in their paper ranges from 95% to 97%. Zhang et al. developed a feature selection algorithm which pre-filters most of the features and further uses a wrapper method to select the best features for a specific classifier [Zhang et al., 2012]. Their approach was evaluated using three classifiers from the traces captured from different networks, achieving more than 94% flow accuracy and 80% byte accuracy on average.

Fahad et al. proposed a method for identifying both optimal and stable features relying on a multi-criterion fusion-based feature selection technique [Fahad et al., 2014]. They used traffic dataset collected from the University of Cambridge, classifying twelve applications with five classifiers (K-Nearest Neighbours, Naïve Bayes, Decision Tree, Support Vector Machine, and Logistic Regression), and getting an accuracy ranging from 70% to 97%. Moreover, surveys recently published [Dhote et al., 2015, Srivastava et al., 2016] report the existing techniques for traffic classification using ML.

3.3 Chapter Remarks

In this chapter, we reported the advantages and issues that emerged with SDN, summarizing their use in proposals for optimizing applications. We realized that these approaches speed up specific applications by modifying network forwarding or by applying QoS or provide higher-level abstractions to make it easier for the user to perform these operations. Some efforts also combat the issues included by SDN by condensing the number of matching entries in the switches' flow tables and by installing the necessary rules in advance.

We also saw some efforts for characterizing the communication patterns through some information such as message type, destination, frequency distributions, and size. We presented the concept of dwarfs for classifying the communication patterns expressed by HPC applications. We related their communication requirements, mapping the application areas and exposing the importance of dwarfs in several computational areas.

We investigated how the communication patterns can be expressed through traffic matrices, considering their spatial and temporal behaviors. We concluded that these applications tend to transmit the same amount of data across the same computing nodes and we call this as a "well-behaved" communication pattern. Computational approaches for classifying the communications were also described in this chapter.

Despite the great diversity of published works using SDN for optimizing applications, to the best of our knowledge, no study exploit the well-behaved communication patterns expressed by the HPC applications for reprogramming the network, aiming to accelerate the applications. So, next chapter we present our first attempt for using the communication patterns to optimize HPC applications.

Chapter 4

Communication Patterns for Programming the Network

As reported previously in Section 3.2, Asanovich et al. investigated the requirements of point-to-point and collective communications, concluding that one possible solution was to reconfigure the network according to the application's communication topology [Asanovic et al., 2006]. In this chapter, we present the Communication Pattern Network Programming (CPNP) framework, our proposal for using the communication patterns expressed by HPC application as the key logic for programming the SDN-enabled devices; it was based on our previously published work [Trois et al., 2017].

This framework keeps a database of communication patterns, annotated with constraints on latency-sensitive and bandwidth-intensive communications. When CPNP receives a call to its API, it identifies the pattern in its database, and evenly places the communications according to existing constraints: latency-sensitive are forwarded through low-latency paths and bandwidth-intensive are spread over multiple links.

CPNP was designed for preventing both issues shown in Table 3.1; it decreases the *(I1) latency for populating the flow tables* by proactively installing all necessary rules before starting the application and *(I2) flow table lookup time* is reduced by grouping the matching flows. So, the first part of our evaluation, we performed experiments for testing these two aspects. These results were published by us in another paper [Trois et al., 2016a].

As most of HPC applications are implemented using the MPI library, the second part of our evaluation we used the OSU Micro-Benchmarks (OMB)[1] for characterizing how the path lenght can influence communications throughput and latency.

In the final part of the evaluation, we made experiments focused on demonstrating the effectiveness of our approach for accelerating HPC applications. We performed tests on two HPC benchmarks and also in HPC applications, comparing CPNP with traditional protocols used for load-balancing multiple-path networks.

4.1 Communication Pattern Network Programming Framework

CPNP is a framework for programming SDN according to the spatial and temporal behaviors expressed by HPC applications; it uses the well-behaved communication patterns

[1]The Ohio State University Micro-Benchmarks are available at: `http://mvapich.cse.ohio-state.edu/benchmarks/`

expressed by these applications as basis for balancing their flows through the available network paths. It is composed of two main components, the Network Programming Module (NPM) and the Traffic Matrix Database (TMD). Figure 4.1 shows the CPNP framework architectural components and their interactions.

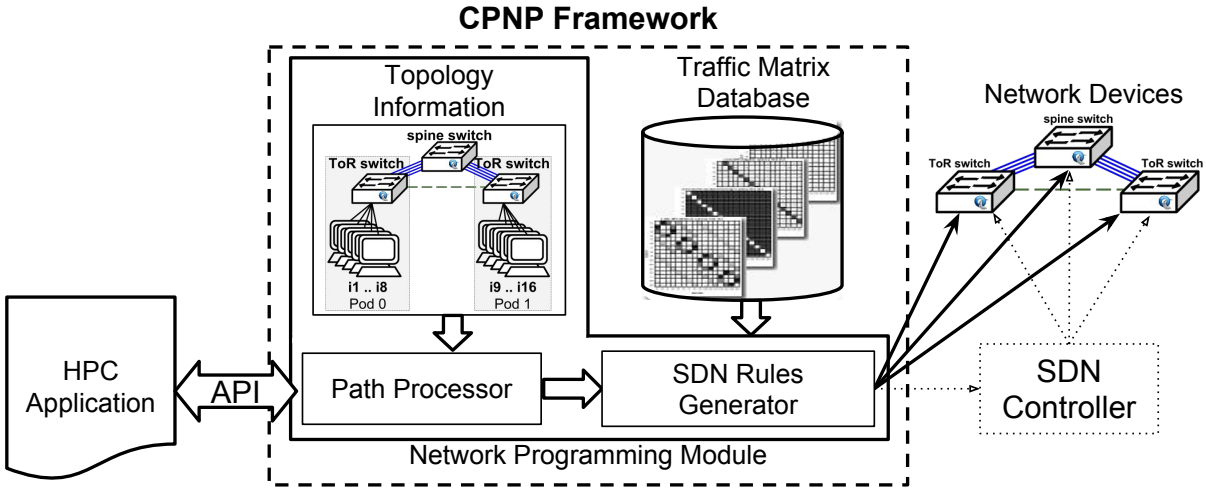


Figure 4.1: CPNP architecture.

4.1.1 Traffic Matrix Database

The TMD stores the computing nodes communication as TMs, where each cell keeps the amount of traffic (packets and bytes) transmitted among every pair of ingress and egress nodes. CPNP provides an option for measuring and recording the communication patterns; in this operation, the switches flow table statistics are collected and based on source and destination addresses it computes the number of transmitted bytes and packets, storing this information in XML format. The recording operation is performed only once per each different TM; for all the upcoming executions, CPNP uses the information stored in the TMD.

The TM may also keep, in each cell, the communication constraints, allowing the HPC developer to inform which communications are latency-sensitive or bandwidth-intensive. This information may also be set through latency and bandwidth thresholds (in bytes per packet). CPNP classifies the cells as bandwidth-intensive if their average packet size (APS) values ($\frac{total\ number\ of\ bytes}{total\ number\ of\ packets}$) are greater than the defined bandwidth threshold; similarly, the cells with APS values lower than the latency threshold are classified as latency-sensitive. An important remark is that the Ethernet protocol specifies a default of 1500 bytes for its Maximum Transmission Unit (MTU). It means that the largest packet that can be sent is 1500 bytes; when the application sends a bigger message, it is broken into smaller pieces before being transmitted. For this reason, the APS shown in the traffic matrices are always lower than 1500.

We will use the MPI_Allreduce primitive to explain CPNP; this MPI function combines values from all nodes and distributes the result back to all nodes. Figure 4.2 shows an example of XML, generated by CPNP, for storing the TM for this primitive. The communication from *i9* to *i1* is classified as bandwidth-intensive, so the CPNP should forward it through bandwidth-intensive paths, while from node *i9* to node *i2* the communication is classified as latency sensitive and must be allocated through path with a maximum of two hops.

A graphical representation of the MPI_Allreduce TM is shown in Figure 4.3. In this figure, the bandwidth threshold was configured to 1100 and the latency set to 120. The bandwidth-intensive cells are shown in blue (dark gray in B&W), the cells in green (light gray in B&W) are

```
<?xml version="1.0" encoding="UTF-8"?>
<traffic_matrix>
  <matrix id="100" name="MPI_Allreduce">
    ...
    <src_host>i9
      <dst_host>i1
        <num_bytes>42752734</num_bytes>
        <num_packets>35346</num_packets>
        <bandwidth intensive="true"></bandwidth>
      </dst_host>
      <dst_host>i2
        <num_bytes>9394</num_bytes>
        <num_packets>81</num_packets>
        <latency max_num_hops="2"></latency>
      </dst_host>
    ...
    </src_host>
    ...
  </matrix>
</traffic_matrix>
```

Figure 4.2: XML representation of MPI_Allreduce TM with latency and bandwidth constraints.

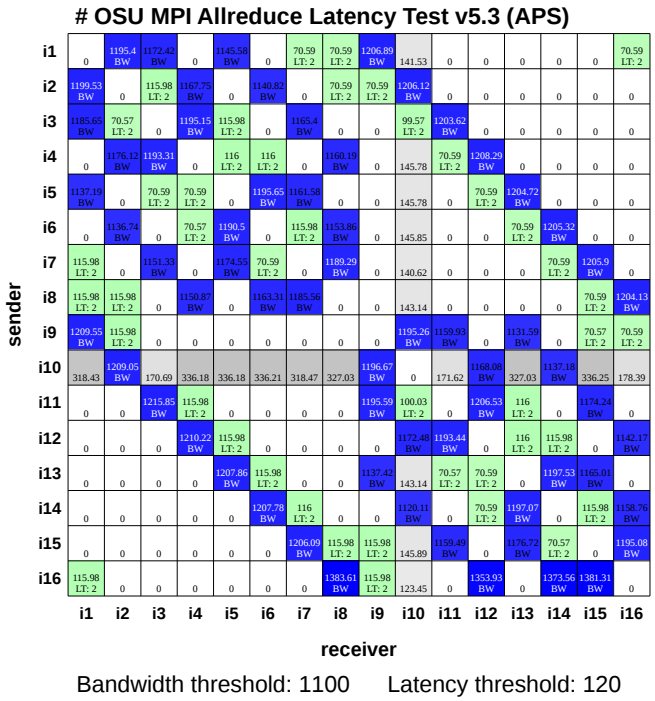


Figure 4.3: MPI_Allreduce TM graphical representation.

defined as latency-sensitive, the gray ones have no constraints. The cells are normalized to the highest value; darker cells indicate higher values and white cells no communication occurred.

4.1.2 Network Programming Module

The NPM is responsible for interpreting the API calls and generating the SDN messages for programming the network devices. It includes the Topology Information Module (TIM), an external API, the Path Processor Module (PPM), and the SDN Rules Generator Module (RGM).

The TIM keeps the topology information such as the network links, the computing nodes locations, and network devices and ports. In this module, the links can also be manually annotated as latency-sensitive and bandwidth-intensive. Although in the current version of CPNP the topology is manually informed, it is possible to obtain and compute this information through the SDN controller or the Link Layer Discovery Protocol (LLDP)[2].

CPNP provides an API allowing the developer to inform which pattern the application is going to communicate. Currently, this API offers a single function (*int net_set_pattern(int pattern_id);*) receiving the *pattern_id* and returning *true* if the network was successfully programmed or *false* on error.

```
...
if(net_set_pattern(100)){ // mpi_allreduce
  MPI_Allreduce(sendbuf, recvbuf, size, MPI_FLOAT, MPI_SUM,
                MPI_COMM_WORLD);
} else {
  printf("Error: could not program the network.");
}
...
```

Figure 4.4: Example of calling CPNP API.

Figure 4.4 shows an example where the API is called to program the network for the *MPI_Allreduce* communication primitive (*pattern_id=100*).

Upon receiving an API call, the NPM uses the argument *pattern_id* to read the TM from TMD, identifying which nodes have exchanged information, as well as their latency and bandwidth constraints. The topology information is acquired from TIM. With this information, NPM allocates the nodes communications among the network paths and sends messages for programming the network devices with these forwarding rules. Due to the well-behaved communication patterns expressed by the HPC applications, and to avoid the time-consuming task of continuously reading the network state, CPNP uses the stored bandwidth and latency information.

PPM first generates low-priority rules for communicating all-to-all computing nodes, preventing any pair of nodes to be incommunicable. For reducing the number of rules in the flow tables, an approach similar to MiceTrap [Trestian et al., 2013] is used, grouping the flows by the destination address. So, mid-priority rules are created using Dijkstra weighted shortest-path [U-Chupala et al., 2014] for allocating the TM cells identified as bandwidth-intensive. Finally, the latency-sensitive communications are accommodated, with the highest priority, on the latency annotated links. Whenever a communication is established in a path, its weight is increased.

After the communication being placed, the RGM creates the OpenFlow messages for proactively programming the network devices' flow tables. In the implemented version, the switches are directly programmed through OpenFlow protocol, but this module can be modified for using any controller's API.

[2] http://www.ieee802.org/1/files/public/docs2002/lldp-protocol-00.pdf

4.2 Evaluation

CPNP aims preventing the latency for populating the switches flow tables and also reduce the flow table lookup time; so, we first evaluated these two aspects. Next, in our evaluation process, we investigated how the communications are affected by the underlying topology, considering latency and bandwidth. Finally, we assessed our approach with HPC applications and benchmarks, aiming to check the overall performance of CPNP on improving their execution times.

The evaluation was structured in three parts, where the first part (Section 4.2.1) is devoted to understanding how CPNP can tackle the SDN issues presented in Section 3.1. The second part (Section 4.2.2), the experiments investigate how the path lenght and message size affect the throughput and latency. Lastly, in Section 4.2.3, we analyze the feasibility of accelerating HPC applications.

For all experiments, the applications were executed 30 times and their execution times were recorded. To avoid the pitfalls introduced by simulation and emulation tools, and be sure that the obtained results are correct and accurate, all experiments were executed in a real testbed. As the baseline, we measured all experiments with the switches configured as L2/L3 mode[3], using the simplest possible topology: all computers connected to a single switch.

Our testbed was composed of 16 Lenovo PCs with processor Intel quad-core 3.2Ghz, 8GB RAM, 1TB HD, 1 Gigabit Ethernet, running Linux Debian 8.2, and MPI implementation mpich-3.2. Three Pica8 P-3290 OpenFlow switches running the operating system PicOS v2.6.4. Each switch has 48 x 1 Gigabit Ethernet ports, four 10 Gigabit optical SFP+ ports, and a Firebolt3 chipset supporting up to 2048 flow entries in its TCAM memory. This switch can operate in two modes of operation: L2/L3 mode and Open vSwitch (OVS) mode. The OVS mode supports OpenFlow 1.4, through Open vSwitch v2.0 integration[4]. The evaluation was performed using OMB v5.3, NPB v3.3.1, and OpenLB v1.0.

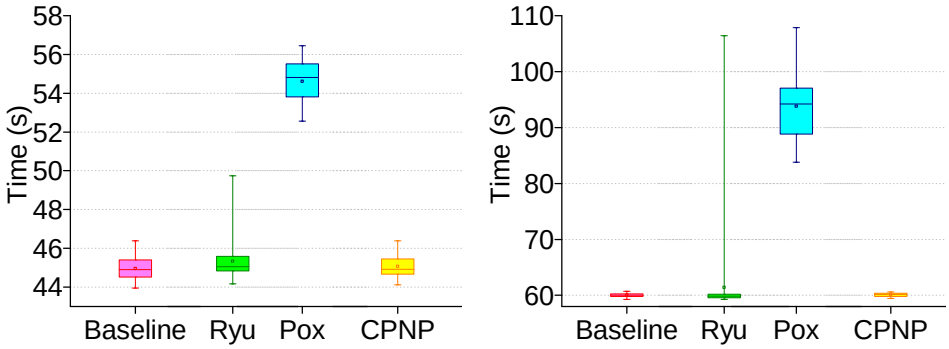
4.2.1 Investigating SDN Issues

For the first part of the evaluation process, we examined how the network programmability and flow load balancing can impact the HPC applications performance. We used the NPB benchmarks that most exchanged information among the computing nodes: *bt*, *cg*, *ft*, and *lu*. These programs were previously described in Section 3.2.1 and their TMs are shown in Figure 3.4. They all presented a single communication phase, meaning that during all execution time their temporal behavior does not significantly change. Their spatial behaviors were recorded and stored in the TMD before performing the tests.

Impact of Network Programmability

To understand the impact of network programmability, we used a single switch programmed with CPNP, Ryu[5], and Pox[6], two well-known SDN controllers, forwarding the flows with their default reactive learning switch. We executed the most rule-intensive applications, *ft* and *lu*, using 16 computers, measuring their execution time and investigating the installed matching rules. Figure 4.5(a) shows the execution times for *lu* application.

[3]Layer 2 / Layer 3: The switch runs as a non-SDN switch.
[4]`http://openvswitch.org/`
[5]`https://osrg.github.io/ryu/`
[6]`https://github.com/noxrepo/pox`



(a) *lu* execution time. (b) *ft* execution time.

Figure 4.5: Time for executing *ft* and *lu* in 16 computers connected to a single switch.

When the switch is programmed with Ryu controller, the application execution time was close to the baseline. However, the first execution the time was higher due to the time for querying the controller. When controlled by Pox, the application took longer to execute, this is explained because the controller is creating and installing rules for every new flow (microflow). Furthermore, Pox installs the rules using an expiration timeout. When these timeouts expire, the rules are removed and the controller is queried again. With CPNP, the execution time achieves the baseline, because it proactively installs the necessary rules.

Figure 4.5(b) shows the measured times for *ft*. When controlled with Ryu the application execution average time was 1.4 seconds slower. However, in the first execution, to install the all-to-all nodes matching rules, it took 106.4 seconds (77% longer than the baseline). When the switch was controlled by Pox, the *ft* execution time was much higher, taking an average of approximately 94 seconds to finish its execution. The average application execution time using CPNP was again near to the baseline.

We note that the Ryu installs the matching rules using source and destination MAC addresses. So, for the *ft* application, it installed 240 rules on the switch's flow table. On the other hand, Pox ranged from 117 to 334 rules. The higher number is because Pox creates microflow rules, and the idle and hard timeouts were responsible for the variation. Due to the all-to-all intensive communication, for this application we defined no latency or bandwidth constraints so, the number of rules installed by CPNP was 16 because they match only the destination addresses.

An important remark is scalability regarding network states for installing rules based on the {source, destination} tuple. It implies that the necessary number of rules grows exponentially, and can be calculated as $n \times (n-1)$, where n is the number of nodes. Considering 48 computing nodes connected to all P-3290 Ethernet ports, and a controller installing an all-to-all communication flows, it will be necessary a total of 2256 flow entries, exceeding the 2048 entries TCAM size. In our testbed switches we observed that the RTT for a ping message goes from 0.3ms for rules stored in TCAM to 4ms when they are located the software flow table. The throughput goes from 936Mbits/s when rules are stored in TCAM to only 4Mbits/s when stored in the software table.

Impact of Load Balancing

For testing the impact of how the flows are allocated through the network paths, we used the two applications that more exchanged traffic considering the pair of nodes (*cg* and *bt*). They were executed in 16 computers connected through topology shown in Figure 4.6. The topology is composed of three switches, one spine and two top-of-rack (ToR) switches. The ToR

switches have eight computers connected to its Gigabit Ethernet ports, and they are connected to the spine with four Gigabit links, annotated in CPNP as bandwidth-intensive.

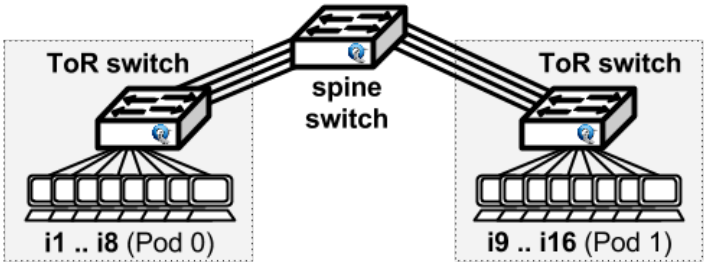
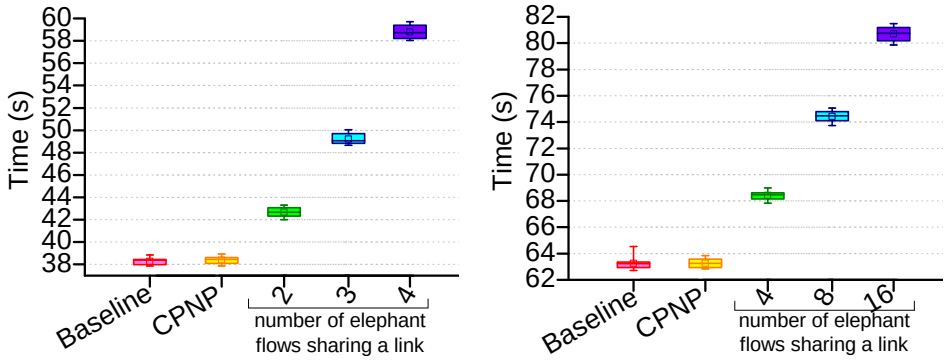


Figure 4.6: Real testbed topology.

To assess the outcome, we compared the application execution time executed in the given topology, programmed with CPNP, against a single switch in L2/L3 mode (baseline). We also measured the execution time when the applications flows were unbalanced on the links.



(a) *cg* execution time.

(b) *bt* execution time.

Figure 4.7: Time for executing *cg* and *bt* in 16 computers connected to the given topology.

Figure 4.7(a) shows the measured execution time of this experiment for *cg* application while Figure 4.7(b) gives the *bt* application. For both applications, the execution time had a very small increase, in average 100 milliseconds, when the network was programmed with CPNP, compared to the baseline; this is explained by the overhead imposed by the topology. However, when elephant flows [Curtis et al., 2011] were allocated sharing a link, the execution time increased considerably.

These two first experiments shown that the network programming may affect the HPC applications performance, as well as, a properly flow allocation can also speed them up. Next we present the second part of our evaluation, where the experiments were carried out for testing the impact of path lenght on communications.

4.2.2 Characterizing the Impact of Path Length on Communications

For understanding the impact of the network topology on throughput and latency for HPC applications, we investigated the MPI point-to-point communication primitives tests available on OMB[7]. We observed the variations in latency and throughput when the traffic was forwarded through paths varying from one to six hops. We chose this maximum value because

[7] `http://mvapich.cse.ohio-state.edu/benchmarks/`

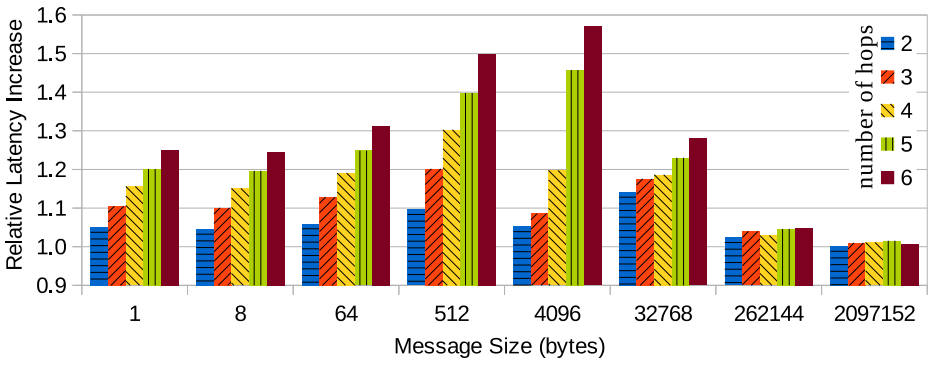
the current datacenter network topologies are designed with a maximum of six hops between any pair of nodes [Yao et al., 2014].

We executed all point-to-point latency tests (*osu_latency*, *osu_latency_mt*, and *osu_multi_lat*), but due to the similarity of the results, we only present the values of *osu_latency* test. This test is carried out in a ping-pong fashion; the sender sends a message with a certain data size (which is increased in each testing step) to the receiver and waits for a reply from it. The receiver receives the message from the sender and sends back a reply with the same data size.

Latency (us)	1	8	64	512	4096	32768	262144	2097152
	83.26	83.63	84.92	88.32	125.67	331.40	2475.44	18239.08

Message Size (bytes)

(a) Latency measured for a single hop.



(b) Relative latency increase.

Number of Hops	1	8	64	512	4096	32768	262144	2097152
2	5.0	4.6	6.0	9.7	5.2	14.1	2.4	0.1
3	10.5	10.1	12.9	20.1	8.6	17.6	4.1	1.0
4	15.6	15.1	19.1	30.3	19.7	18.7	2.9	1.1
5	20.1	19.6	24.9	39.8	45.7	23.0	4.6	1.6
6	25.1	24.6	31.3	49.8	57.0	28.0	4.8	0.1

Message Size (bytes)

(c) Percentage increase in latency.

Figure 4.8: Latency for different hop count and message size.

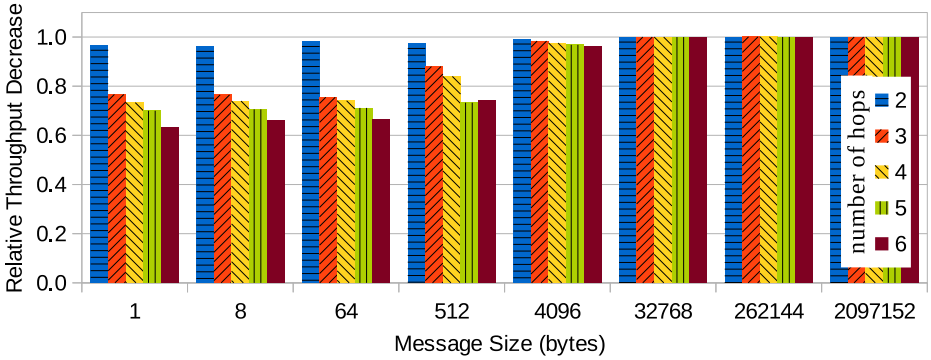
Figure 4.8(a) shows the latency values (in micro-seconds) for the different tested message sizes when the communication is performed through a single hop (only one switch). Figure 4.8(b) shows the relative increase in latency, compared to the single hop value, varying the number of hops from two to six.

Figure 4.8(c) shows the percentage increase in latency, where the cells in red represent an increment higher than 10%. This figure facilitates the increased latency visualization; for example, considering an application with latency-sensitive communication, tolerating an increase up to 10% in latency, and transmitting messages with 512 bytes. In this case, the application traffic must be forwarded through a path with a maximum of two hops.

Just as happened with the latency tests, the values obtained for all bandwidth tests were similar, so we show only the results of OMB Bandwidth Test (*osu_bw*). This test has a sender sending out a fixed number of end-to-end messages and waiting for a reply from a receiver. The receiver sends the reply only after receiving all these messages. The bandwidth is calculated based on the elapsed time and the number of bytes sent by the sender.

Throughput (Mb/s)	2.4	19	151	559	863	914	930	933
Message Size (bytes)	1	8	64	512	4096	32768	262144	2097152

(a) Throughput values for a single hop.



(b) Relative throughput decrease.

Number of Hops	1	8	64	512	4096	32768	262144	2097152
2	3.3	3.8	1.8	2.6	0.8	0.0	0.0	0.0
3	23.3	23.5	24.5	12.0	1.6	0.0	0.0	0.0
4	26.7	26.1	25.9	16.0	2.4	0.0	0.0	0.0
5	30.0	29.4	28.8	26.7	3.2	0.0	0.0	0.0
6	36.7	34.0	33.5	26.0	3.9	0.0	0.0	0.0
Message Size (bytes)								

(c) Percentage decrease in throughput.

Figure 4.9: Throughput for different hop count and message size.

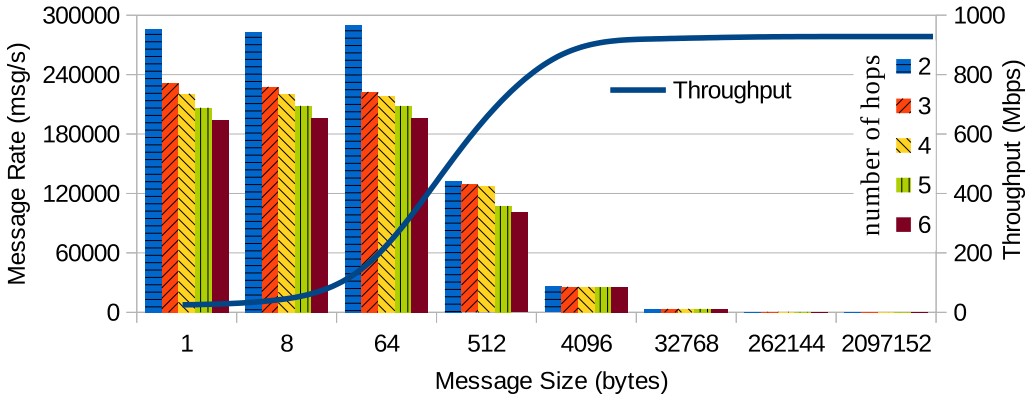
Figure 4.9(a) shows the measured throughput (GB/s) for transmitting the different messages size through one hop and Figure 4.9(b) plots relative decrease in throughput for hop variation. Figure 4.9(c) present the percentage increase compared to the single hop values. For messages larger than 4096 bytes, the link becomes saturated, and throughput stops decreasing. We can also see that for messages larger than 2048 bytes, the throughput percentage increase is lower than 10%, even if they are forwarded through six hops. Thus, we conclude that the number of hops does not significantly affect bandwidth-intensive communications.

We also executed the OMB Multiple Bandwidth / Message Rate test (*osu_mbw_mr*); this test evaluates the aggregate bandwidth and message rate between multiple pairs of processes. Each of the sending processes sends a fixed number of messages back-to-back to the paired receiving process before waiting for a reply from the receiver.

In Figure 4.10(a), it can be seen the message rate (msg/sec) for different message size forwarded through a single hop. There is a large variability on the throughput for small messages

Message Size (bytes)	Message Rate (msg/s)
1	297858
8	297070
64	294350
512	136293
4096	26344
32768	3526
262144	444
2097152	56

(a) Message rate values for a single hop.



(b) Message rate variation chart.

Figure 4.10: Message rate for different hop count and message size.

[1,8,64,512], varying from [2.4,19,151,559] Mbps respectively, as expected. Whereas for larger messages (≥ 4096 bytes), it achieves the capacity of the link, decreasing the message rate variation. Figure 4.10(b) shows that the message rate falls sharply when small messages (≥ 64 bytes) are forwarded through a higher number of hops. It is possible to see that for these messages, through one hop, the message rate is around 300,000 msg/sec; however, when the test is performed via six hops, this rate decreases to less than 200,000 msg/sec. Thus, if the HPC application depends on exchanging a huge number of small messages, then it is mandatory to send them along paths with the least number of hops.

In the coming section, we performed evaluations aiming to enhance the performance of HPC applications.

4.2.3 Improving HPC Applications

For testing the feasibility of improving the performance of HPC applications, we performed three different experiments. In the first experiment, we used CPNP for improving the MPI collective primitives by balancing their communications through the network paths. In the second experiment, our approach is applied for reducing the execution time of NPB, and in the third experiment, we optimized the network for HPC applications implemented using the OpenLB library [Heuveline and Latt, 2007].

For these experiments, we used the testbed topology shown in Figure 4.11; it is composed of one spine and two ToR switches. The sixteen computing nodes (*i1..i16*) are divided between the two ToR switches. The spine switch is connected to the ToR through four links, annotated as bandwidth-intensive (identified in blue). The topology also includes a link connecting the ToR switches; this link is annotated as latency-intensive (in green). Besides the baseline, we also measured these experiments with the network devices configured with LACP for aggregating the Ethernet interfaces into a single logic interface.

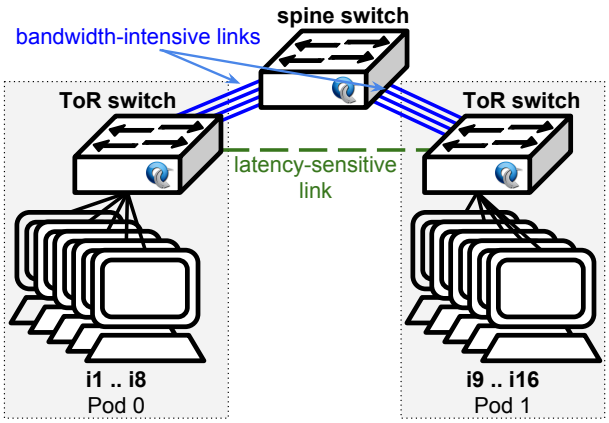


Figure 4.11: Testbed topology annotated with bandwidth and latency information.

The LACP applies a hash function on some packet header fields for choosing the output port to forward packets. The problem is that two or more long-lived flows can collide on their hash and end up on the same output port, creating a bottleneck, overwhelming the switch buffers, and degrading the overall switch performance [Al-Fares et al., 2010].

Improving MPI Collective Communications

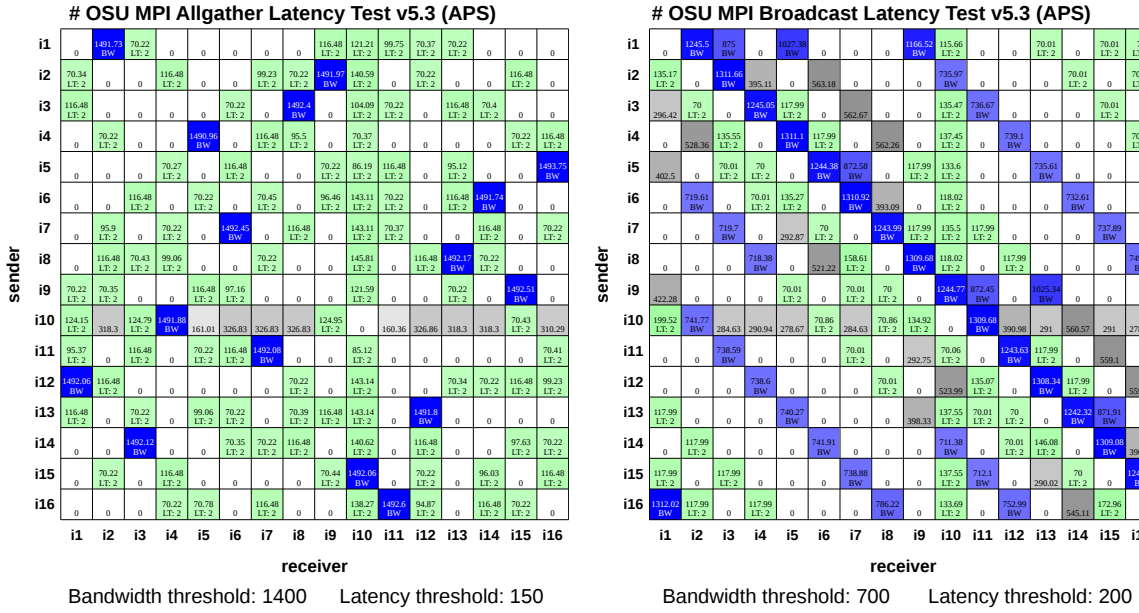
MPI collective primitives are used in most HPC applications, being responsible for a significant fraction of the communication time [Rabenseifner, 1999]. We selected the three OMB tests that most exchanged data through the network to conduct this experiment (MPI_Allgather, MPI_Allreduce, and MPI_Bcast). These benchmarks measure the average latency of collective operations across their processes, for various message lengths, over a large number of iterations. They were executed in our testbed using 16 processes (one process per computer). For increasing the amount of traffic transmitted through the spine switch, when evaluating the primitive MPI_Allgather, the computing nodes were informed in a random order.

Before running CPNP, we measured and stored these benchmarks TMs, annotating some cells as bandwidth-intensive and latency-sensitive, as shown in Figure 4.3, Figure 4.12(a), and Figure 4.12(b).

We tested different thresholds, choosing those that provided the lowest average latency for each test. The allocation of bandwidth-intensive traffic (in blue) had a significant impact on results, meaning that, when they were properly balanced, the benchmarks achieved a latency time close to the baseline. However, for the evaluated MPI primitives, allocating the latency-sensitive communications on the latency link (in green) did not influence the result significantly. This is explained because latency-sensitive communications are dependent on bandwidth-intensive messages.

Figure 4.13(a) shows the OSU MPI_Allreduce latency test normalized to the baseline. It is possible to see that, when the network was configured using LACP, for messages larger than 8192 bytes, there is a significant increase in the average latency, taking nearly 200% longer than the baseline for the largest message. When the network devices were programmed with CPNP, in the worst case, the average latency took 25% more than the baseline.

The Figure 4.13(b) and Figure 4.13(c) present the relative latency increase for MPI_Allgather and MPI_Bcast respectively. Both charts show a considerable increase for message larger than 8192 bytes. In both tests, in the worst case, LACP increased latency time more than twice, while CPNP increased 57% and 17% for MPI_Allgather and MPI_Bcast.



(a) MPI_Allgather (random). (b) MPI_Bcast.

Figure 4.12: MPI collective primitives' TMs annotated with bandwidth and latency constraints.

These results show that a proper allocation of communication is fundamental for reducing the latency, and consequently, reducing the overall execution time of MPI operations. Next section we present our tests with parallel benchmarks and HPC applications.

Parallel Benchmarks

We modified four tests from NPB, including calls to CPNP API, for programming the network according to their communication patterns. Again we selected the benchmarks that most exchanged information among their computation nodes: *bt*, *cg*, *ft*, and *lu*.

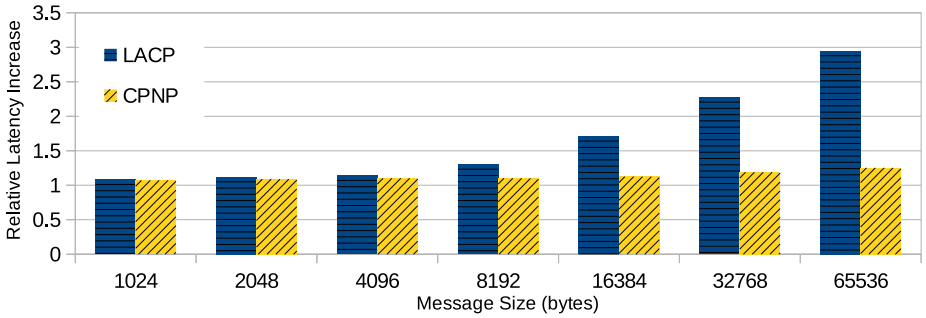
For Figure 4.14 and Figure 4.15, the baseline comprises the computation time coupled with the communication time through a single switch. The overhead imposed by the topology and the manner the communications are balanced on the links by LACP and CPNP are displayed on the top of each bar.

For *cg* and *lu*, the execution time using CPNP was close to the baseline; the increment for *bt* and *ft* was respectively 4% and 36%. When the flows were distributed through LACP, the baseline times for *bt*, *cg*, *ft*, and *lu* incremented respectively 14%, 52%, 161%, and 8%.

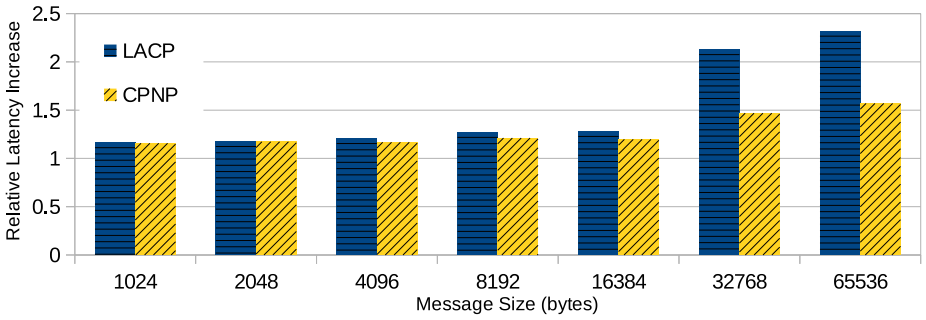
To confirm that the LACP was not equally dividing the communications among the paths, we analyzed the switch ports statistics when running the *ft* kernel and we realized that three ports had transmitted approximately 3 Gigabytes and the other port forwarded merely 2.6 Megabytes. With CPNP, all ports transmitted around 2.2 Gigabytes.

HPC Applications

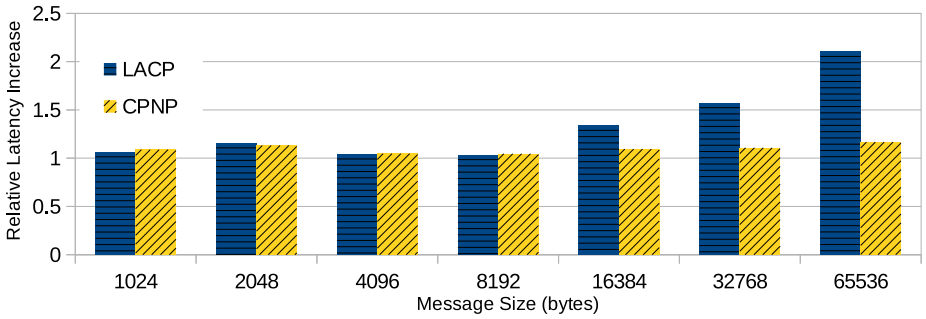
Finally, we modified two HPC applications, including calls to CPNP API, for amending the forwarding according to their communication patterns. The first application examines a steady flow past a 3D cylinder placed in a channel. The cylinder is offset somewhat from the center of the flow to make the steady-state symmetrical flow unstable [Schäfer et al., 1996]. The second application implements a backward facing step, being used to simulate flows through



(a) MPI_Allreduce latency test relative increase.



(b) MPI_Allgather latency test relative increase.



(c) MPI_Bcast latency test relative increase.

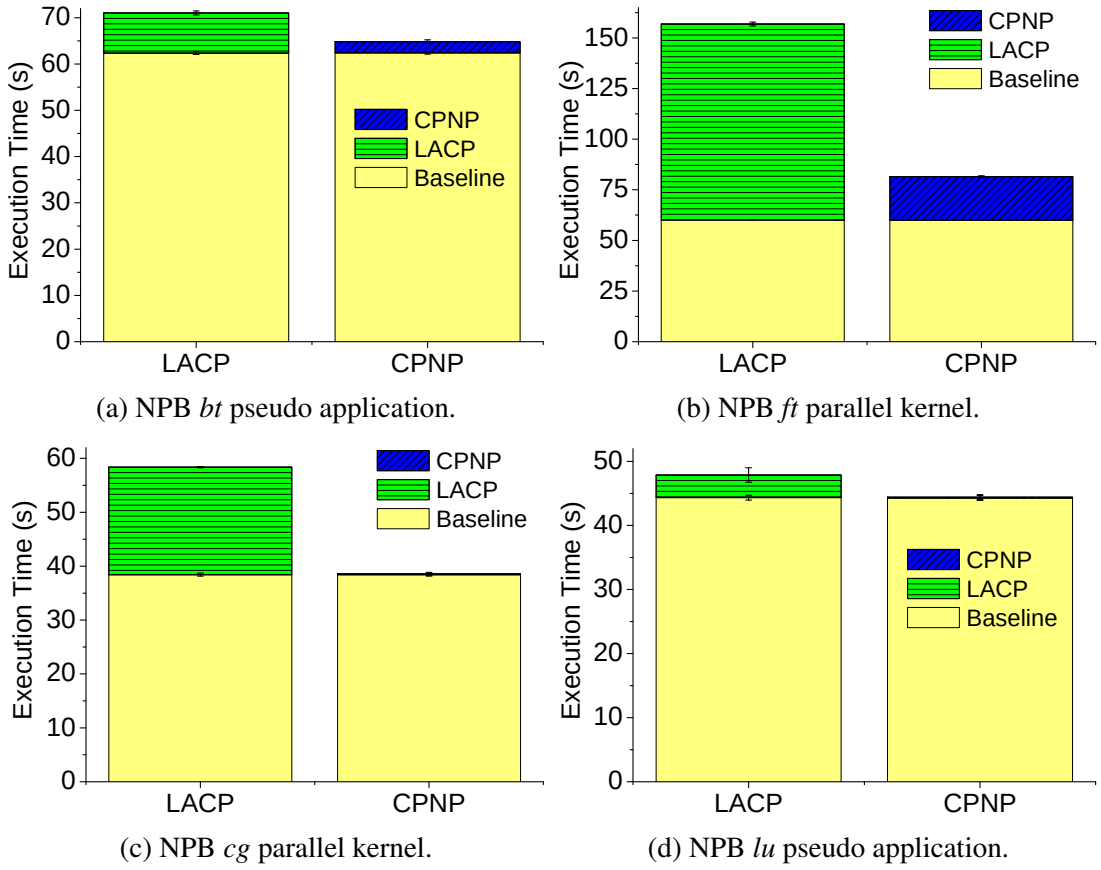
Figure 4.13: OMB MPI collective latency tests.

rough-walled rock fractures [Briggs et al., 2014]. Both applications were implemented using the OpenLB library [Heuveline and Latt, 2007].

The 3D cylinder application had a 8% increase in the execution time when the flows were placed with CPNP and 17% with LACP. By examining the ports of switches, we realize that again LACP did not balance the communications; some ports transmitted about 11 Gigabytes, while other ports only 3.7 Gigabytes. The execution time of the second application increased about 26.5% when the communications were placed with LACP, comparing with CPNP.

For these applications, as well as for the *ft* kernel, even when CPNP properly distributed the communications on the existing links, the overhead was considerable. This happened because the amount of information exchanged between the two Pods was higher than the capacity of existing links. This problem could be mitigated by including new bandwidth links from ToR switches to the spine switch.

We also measured the overhead introduced by CPNP API calls. Among the tested applications, the time for programming the switch's flow tables, on average, was 0.47 seconds. This overhead is slightly higher for applications that need to install more rules. For instance, the *bt* application demanded the installation of 48 rules, which is twice the number of rules



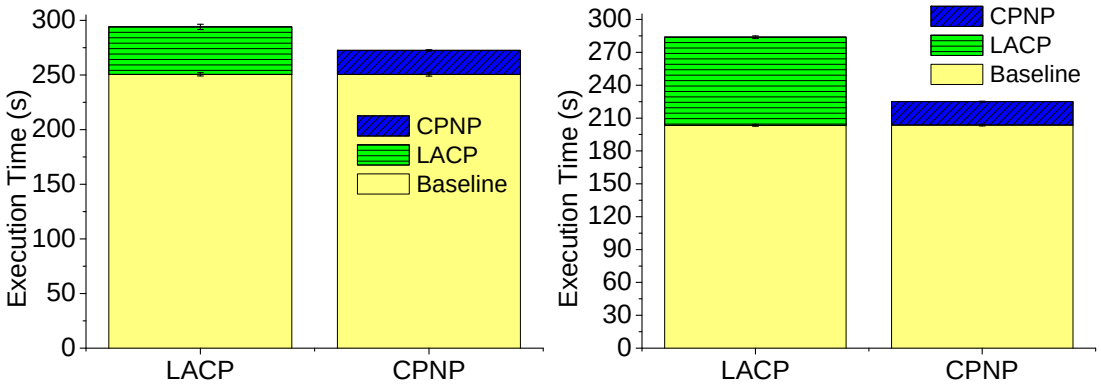
(a) NPB *bt* pseudo application.

(b) NPB *ft* parallel kernel.

(c) NPB *cg* parallel kernel.

(d) NPB *lu* pseudo application.

Figure 4.14: NAS Parallel Benchmarks execution times.



(a) 3D cylinder.

(b) 3D backward facing step.

Figure 4.15: HPC applications execution times.

compared to other applications. The average time for programming the rules for *bt* was around 0.54 seconds.

These experiments show that even in the case of extremely simple topology, HPC applications performance is highly affected by unequal loads in the network paths. However, CPNP provides a way to program the network to meet the pattern communications to be suitably balanced through network paths. Moreover, CPNP allows more agility for setting up routes/flows on the underlying physical network topology, exploring the existing redundant paths for flows that are throughput oriented or providing shortest paths for those that need low latency guarantees.

4.3 Chapter Remarks

In this chapter we presented CPNP, a framework for evenly placing the HPC applications communications on the available network paths. Our approach relies on (i) storing the application traffic matrices annotated with bandwidth and latency constraints; (ii) providing an API to enable the HPC application developer to easily modify the network forwarding for his needs, and (iii) using this information for balancing the communications on the network paths.

We performed experiments for showing that our solution can efficiently handle some issues introduced by SDN. We also studied the impact of path length and message size on latency and throughput, concluding that the number of hops does not significantly affect bandwidth-intensive communications and if the HPC application depends on exchanging a huge number of small messages it is mandatory to send them along paths with the minimum number of hops. Finally, we presented the experiments for demonstrating the effectiveness of our approach by improving the MPI collective operations and accelerating up to 26.5% the execution time of HPC applications.

The CPNP framework ease the complex, tedious, and error-prone task of programming the network, but it still requires HPC application to be modified by including calls to its API. So, in the next chapter, we present an novel method for automatically classifying the communication patterns, reporting how the CPNP framework was extended for including this method.

Chapter 5

Abstracting and Automating the Network Programming

In the previous chapter we proved through CPNP that the performance of HPC applications can be improved when their flows are balanced across the existing network paths. The problem is that CPNP requires the application to be modified by including calls to its API so, an automated solution would be required for automatically detecting the communication patterns.

The first challenge for developing an automated solution was to detect which application is currently using the network resources. As reported in Section 3.2.2, the existing methodologies for identifying applications are port-based, DPI, and ML-based. The first two methods have issues and restrictions and, as the goal of ML is to develop methods that can automatically detect patterns in data, we decided using ML in our implementation.

As far as we know, the traffic classification employing ML mainly uses statistical information collected from the network for feeding the ML classifiers. As CPNP was already acquiring the TMs, we developed a novel method employing characteristics (features) extracted from the application TMs, as an input for the classifiers. This method works by applying a function for "unscrambling" the TMs, resulting in the rendering of different visual textures for each communication pattern. Then, we used two well-known textural representations, Uniform LBP (ULBP) [Ojala et al., 2002] and Robust LBP (RLBP) [Zhao et al., 2013] for extracting the feature vectors that feed two different classifiers, Support Vector Machines (SVM) [Vapnik, 1999] and Random Forest (RF) [Breiman, 2001]. We evaluated the ML method on two real testbeds, realizing that it correctly classified more than 98% of communication patterns, while other proposals reported in the current bibliography obtained, at best, an accuracy of 87%.

In this chapter we present this ML method and a framework, derived from the CPNP, called TReco (**T**raffic Matrix **Reco**gnition framework). This framework joins the ML method for automatically identifying the applications with our approach for programming the network according to the communication patterns. In our evaluation the execution time was reduced by 15.8% when the framework was used to tune the network according to the communication patterns.

5.1 Traffic Matrix Recognition Framework

The **T**raffic Matrix **Reco**gnition (TReco) framework was designed to identify which HPC application is using the network and to reprogram the switches for optimizing the network according to the application's communication pattern.

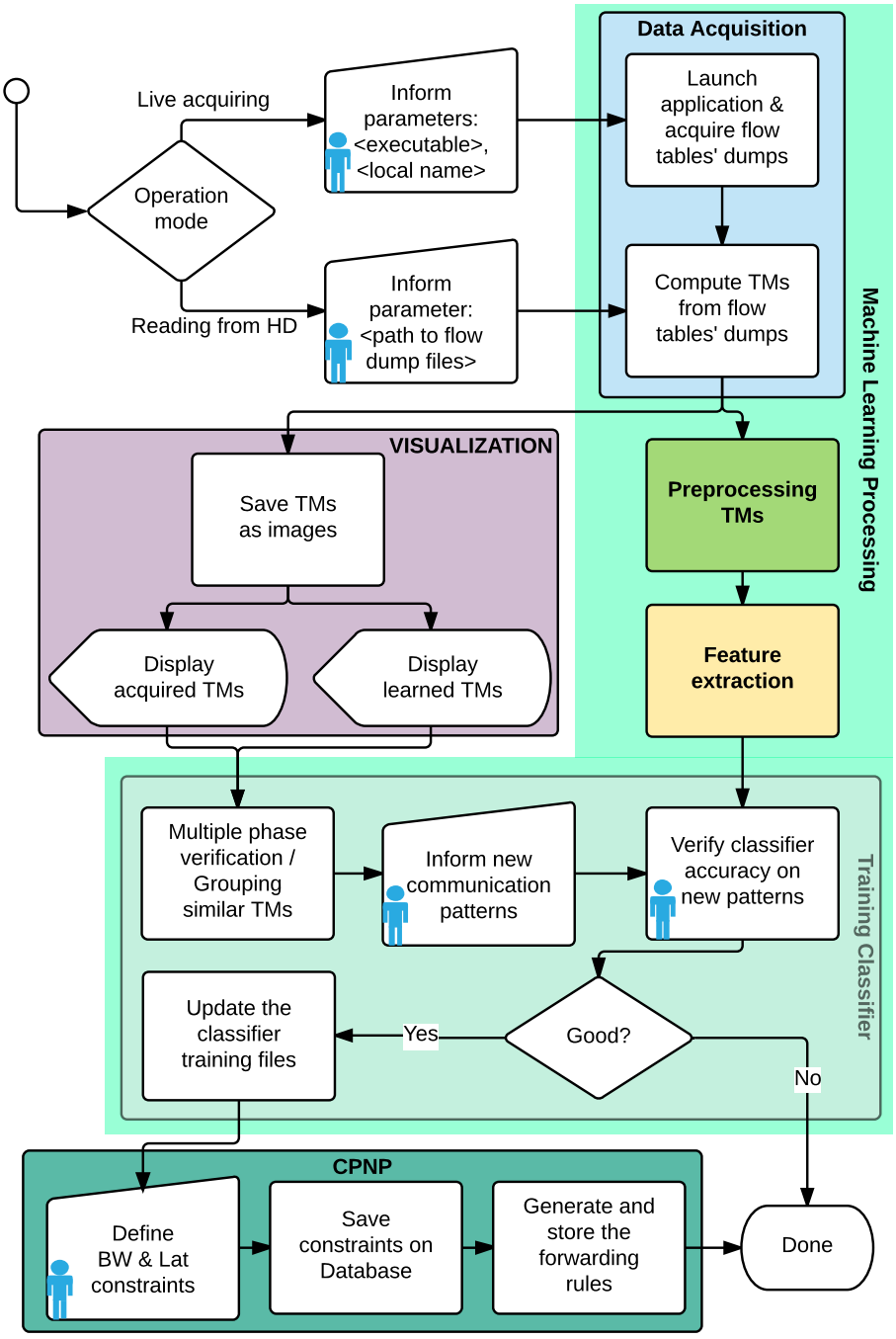


Figure 5.1: Flowchart for learning the communication patterns.

TReco has two operation modes, the first mode is used for "learning" the communication patterns. The Figure 5.1 shows the flowchart of this mode; the ML processing is displayed on the right side, highlighted in light green, and the stages where user intervention is required are marked with the drawing of little man.

The first stage consists in the acquisition of TMs. It can be done by live acquiring the TM from the network; in this case, the user informs the application executable and a name used for storing locally the TMs. The second option is reading files containing TMs previously acquired from the network.

Next, two processes occur in parallel, (i) all acquired TMs are displayed to the user that identifies the communication patterns, grouping the TMs in those patterns. (ii) For preprocessing

the TMs, we developed a method for generating different textures for each communication pattern, then a visual descriptor method is applied for extracting the feature vectors that are going to be used as input by the ML classifier.

To be sure that the amount of TM acquired is sufficient, and that the TReco will be able to classify them in the running mode, an accuracy performance verification is executed. In this stage, the new communication patterns are used, along with the previously learned patterns, for feeding the classifier, which identifies them and shows to the user the prediction accuracy. If the user considers the results adequate, the classifier training files are updated with the learned patterns.

Lastly, the user informs the bandwidth and latency constraints for the learned communication patterns, this information is stored in the "CPNP part of the framework", and used for generating the forwarding rules.

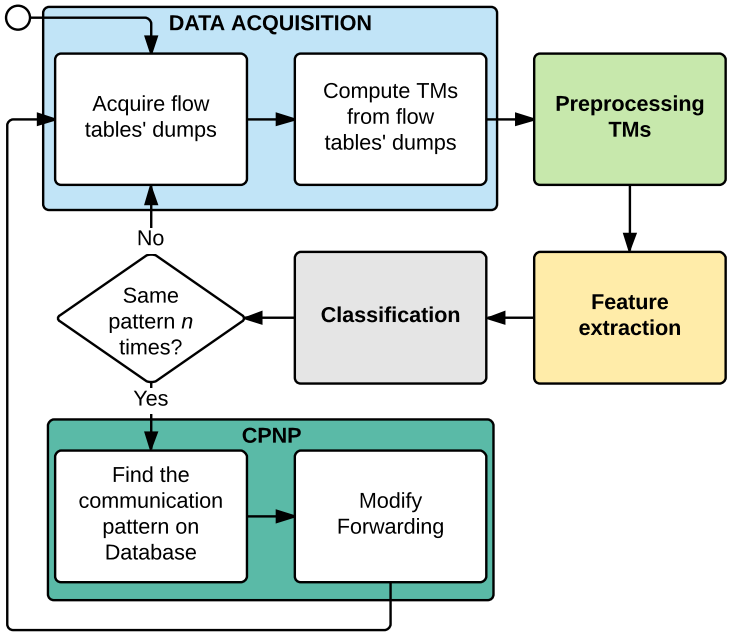


Figure 5.2: Flowchart for classifying the communication and modifying the forwarding.

The second operating mode of TReco is the running mode; in this case, the system continuously acquires the TMs on-the-fly, applies our preprocessing function, extracts the feature vectors, and classifies the communication pattern. When the same communication pattern is identified a predetermined number of times, the system modifies the switches' forwarding tables to optimize the network for that pattern. The flowchart of this mode is shown in Figure 5.2. The processing stages displayed in the flowcharts are described in the next sections.

5.1.1 Data Acquisition: Getting the Traffic Matrices

The first stage of a pattern recognition system is the data acquisition. In our case, the input depends on collecting the applications' TMs. As stated in Section 3.2.1, a TM can be formalized as a matrix M, where each position $M_{[i][j]}$ holds the number of bytes transmitted during a time t, from node i to node j. Also, the temporal behavior during the application execution can be seen as a set of multiple TMs $M_{[i][j]}(t)$, collected for every time t.

An important remark is that the manner of TMs are collected is orthogonal to TReco, which means that it might be obtained using different techniques [Gong et al., 2015,

Trois et al., 2016a]. We developed two ways for collecting the TMs, the first way was described in our previous work [Trois et al., 2017], it consists in installing traffic measurement rules in each SDN-enabled switch, and at every time t, the switches' flow tables are dumped to files, finally these files are parsed for generating the TMs.

The second way was using the *strace* Linux utility for intercepting all the network related system calls. These system calls are recorded to a file and parsed locally in each computing node, extracting the total amount of bytes transmitted to and received from all other nodes. At an adjustable time t, all nodes sent the parsed data to a master node which consolidated it into the TM for the time t.

5.1.2 Preprocessing: The Unscrambling Function

Usually, the collected data for pattern recognition cannot be easily processed by computer algorithms and some preprocessing must be performed for facilitating the classification.

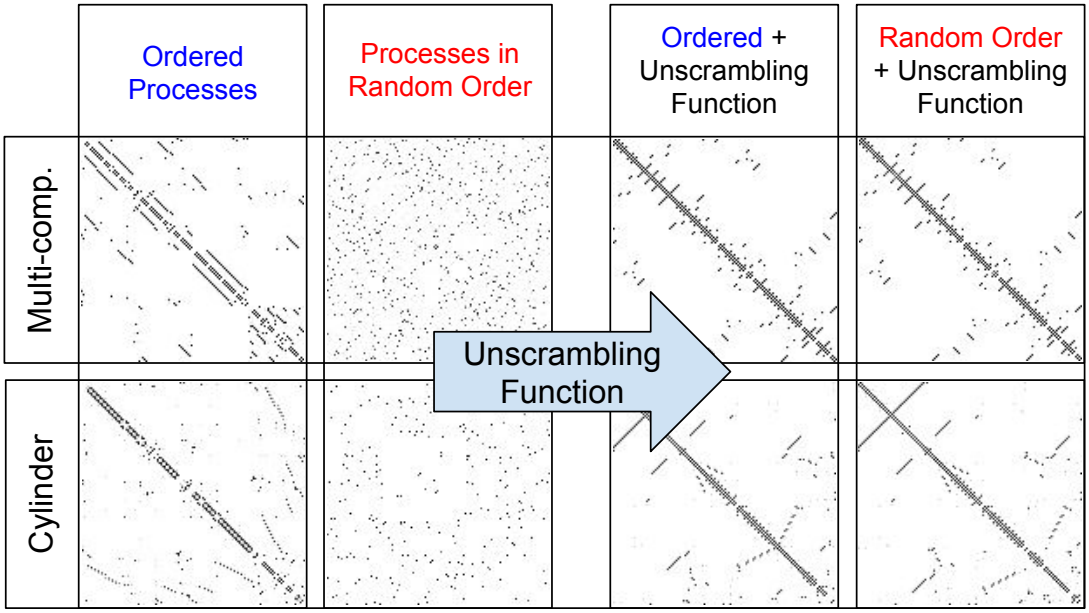


Figure 5.3: Unscrambling function.

Commonly, when the users want to execute an application in a HPC environment, they have to submit it to a job tracker or queuing system. This system is responsible for automatically allocating the required computing resources and returning a list of computing nodes (cn_1, cn_2, ..., cn_n). Figure 5.3 shows TMs when the application's processes (ap_1, ap_2, ..., ap_n) are allocated in an ordered manner (i.e. ap_1 in cn_1, ap_2 in cn_2, ..., ap_n in cn_n) and when the processes are randomly distributed on the computing nodes ("*Hosts in Random Order*" column). If the processes are allocated in a random order, it is not possible to identify any visual textures in their TMs, because they are composed of many scattered points.

For tackling this problem, we have developed an *Unscrambling Function* that reorders the TM lines and columns, rendering a different texture for each communication pattern. The basic premise of this function is, for each line, to "bring" the most communicating pair of nodes close to the matrix main diagonal. As we can see, in the two right columns of Figure 5.3, after applying the Unscrambling Function, it is possible to visually distinguish the applications looking at the generated textures, independently of the hosts allocation order.

The Pseudocode 1 presents the unscrambling function. For every line in the TM, it first locates the column with the highest value (after the main diagonal), and then swaps the entire column, bringing it close to the main diagonal. As the computing nodes are the x and y-axes of the TM, it is mandatory also to swap the column situated in the same position of the swapped line.

Algorithm 1 The unscrambling function

```
1:  function UnscrambleMatrix(mat)
2:      max ← 0
3:      max_pos ← −1
4:      for line ← 0, sizeof(mat) do
5:          for col ← (line + 1), sizeof(mat) do
6:    // find the position of the highest value for each line
7:              if mat[line][col] > max then
8:                  max ← mat[line][col]
9:                  max_pos ← col
10:             end if
11:         end for
12:         if max_pos > 0 then
13:             for i ← 0, sizeof(mat) do
14:                 aux = mat[i][line + 1]    // swap column
15:                 mat[i][line + 1] = mat[i][max_pos]
16:                 mat[i][max_pos] = aux
17:                 aux = mat[line + 1][i]    // swap line
18:                 mat[line + 1][i] = mat[max_pos][i]
19:                 mat[max_pos][i] = aux
20:             end for
21:         end if
22:     end for
23: end function
```

5.1.3 Feature Extraction

After collecting and unscrambling the TMs, the next stage consists of extracting the feature vector. As stated before, our proposal considers the textures, so we applied the Local Binary Pattern (LBP) [Ojala et al., 2002], a model widely used for texture classification.

For doing the LBP, we consider C as each cell in the TM. The value of C is compared with its eight neighbors cells, starting with the top-left, and following a clockwise order. If C is greater than the neighbor's value, write "0", otherwise write "1", resulting in an 8-digit binary number which is converted to decimal; this is the LBP value calculated for that cell. The same operation is performed with all cells in the TM and a histogram is created as a 256-dimensional feature vector.

We implemented two LBP optimizations, the first, called ULBP [Ojala et al., 2002], reduces the length of the histogram to a 59-dimensional feature vector. This optimization introduced a concept based on the transition between 0's and 1's in the LBP image. A binary LBP code is considered uniform if the number of transitions is less than or equal to 2, also considering that the code is seen as a circular list. That is, the code 11010011 is not considered uniform because it contains four transitions. But the code 11110011 is characterized as uniform because it has only two transitions.

The second is named RLBP [Zhao et al., 2013], which considers the input image as being noisy and proposes changing one single bit from the original LBP (only if this modification turns it in a uniform pattern). In the previous example, if we change the third bit from 0 to 1

(11**1**10011), it will result in a uniform pattern, which is a more meaningful pattern for the texture representation and classification.

5.1.4 Traffic Matrix Classification

As our assumption was to classify the applications based on already known TMs, so we used the supervised classification. In this type of ML algorithm, a labeled database, defined as the training set, is used as the input to the classifier. We used the classifiers SVM [Vapnik, 1999] and RF [Breiman, 2001] to evaluate the discriminative power of our proposed representation.

SVM is a popular classification algorithm, which builds a hyperplane in a high-dimensional space that may be used either for classification or regression. Different from other linear discriminant functions, it provides the optimal hyperplane that separates two classes. The RF is an ensemble approach that uses decision tree predictors. The rationale behind ensemble methods is that a group of weak learners can come together to form a strong learner.

5.2 Evaluation

For evaluating our proposal, we first investigated the accuracy our method for characterizing the applications by using the traffic matrices' visual features. Next, we investigated the processing time of each stage of our approach and lastly, we analyzed its overall performance for reducing the execution time of HPC applications.

First, for assessing our proposed ML approach, we used two different scenarios, an Intel based cluster and a HPC machine, executing two different types of applications that are normally executed in HPC environments: MapReduce and scientific applications. We measured the accuracy of our method with two classifiers and compared it with four relevant ML methodologies found in current literature. The second evaluation step aimed to measure the overall performance of our proposal. We first investigated the amount of time required for each processing stage of the framework. Next, we executed a queue of applications, letting TReco for automatically identifying the running application and reprogramming the network for its communication pattern. We measured the execution time for the entire queue with our approach and with LACP.

5.2.1 Experimental Testbed

Intel Cluster (Cluster A) This cluster is composed of 16 Lenovo PCs each with Intel 8-core 3.2Ghz processors, 8GB RAM, 1TB HD, 1 Gigabit Ethernet, running Linux Debian 8.2, mpich-3.2, Disco MapReduce[1] (v0.5.4),NPB v3.3.1, and OpenLB v1.0. These computers are connected to a 48 port Gigabit Ethernet Pica8 P-3290 switch running the operating system PicOS v2.6.4. This switch supports OpenFlow v1.4 through Open vSwitch[2] (v2.0) integration.

BlueCrystal Phase 3 (Cluster B) This cluster is a HPC machine belonging to the University of Bristol[3] which is comprised of 223 base blades, where each blade has 2.6 GHz SandyBridge processor with 16 cores, 64GB RAM, and a 1TB SATA disk. Besides these "base blades," there are also 100 blades that can host dual GPGPUs, and 18 large memory blades each containing 256GB of memory. This cluster runs Scientific Linux (v6.4), Disco MapReduce

[1] http://discoproject.org/
[2] http://openvswitch.org/
[3] https://www.acrc.bris.ac.uk/acrc/phase3.htm

(v0.5.4), OpenLB v1.0, Torque (v4.2.4.1)[4] plus Moab (v7.2.9)[5] as the queuing system, and the mpich2 (v1.4.1p1)[6] as the standard MPI.

Experimental Datasets

To evaluate the classification method, we ran the MapReduce and scientific applications, performing tests with different number of processes (16, 32, 64, and 128), collecting their TMs every second for a period of 30 minutes. For comparing TReco with the previous version of the framework, we repeated the experiments reported in Section 4.2.3, executing the NPB with 16 and 64 processes. [7].

MapReduce Is often used to solve problems when a vast amount of input information can be processed concurrently with a large number of computers (nodes). Usually, MapReduce takes advantage of data locality, processing it on or near the storage assets. In this computation phase, no communication occurred, and the collected TMs were "blank".

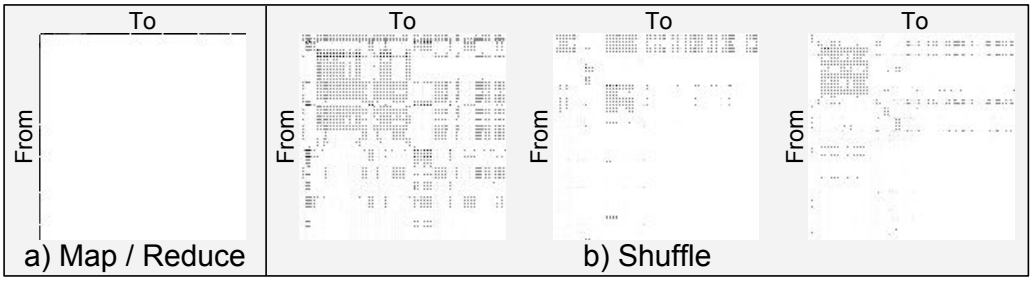


Figure 5.4: MapReduce traffic matrices.

To evaluate MapReduce, we executed the Disco application examples[8] and collected their TMs. After gathering the TMs, we analyzed them and we could identify two different communication patterns, which are presented in Figure 5.4. The operations Map and Reduce have a similar communication pattern (Figure 5.4(a)) where the only communication that exists is between the master and all other nodes and vice-versa. On the other hand, the data shuffling operation is more network intensive. As it can seen in Figure 5.4(b), this operation does not have a singular pattern and the communication may occur among many pairs of random nodes.

Scientific Applications In our evaluation, we measured the TMs of the four most network-intensive scientific applications implemented with the OpenLB library: a) **Multi-component** demonstrates the instability generated by a heavy fluid penetrating a light one; b) **Cylinder** simulates flow passing through a circular cylinder; c) **Bifurcation** reproduces human lungs bifurcation; and d) **Poiseuille** law that relates the pressure drop of a steady flow through a long cylindrical pipe with a constant radius. Figure 5.5 shows the TMs acquired from these applications.

All matrices presented in Figure 5.4 and Figure 5.5 have been preprocessed, meaning that the unscrambling function has already been applied to them.

[4] `http://www.adaptivecomputing.com/products/open-source/torque/`
[5] `http://www.adaptivecomputing.com/products/hpc-products/moab-lite/`
[6] `https://www.mpich.org/`
[7] This dataset is publicly available at: `https://github.com/celiotrois/treco_dataset`
[8] `https://github.com/discoproject/disco/tree/develop/examples/`

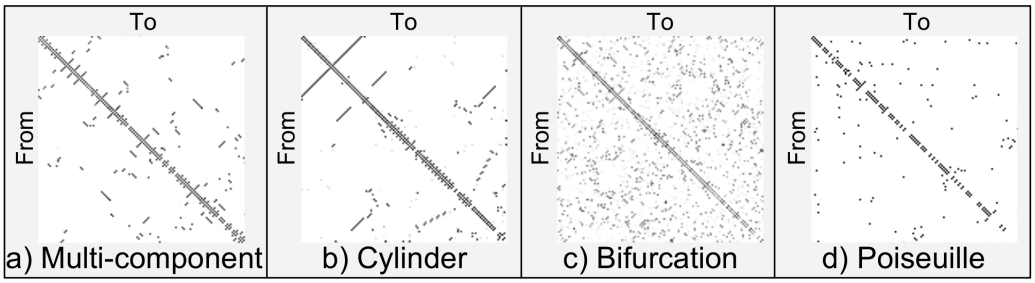


Figure 5.5: Scientific applications traffic matrices.

Table 5.1: Feature vectors used in related works.

Reference	**Feature vector**
Eerman et al. [Eerman et al., 2006]	total number of packets average packet length (client to server) average packet length (server to client) average packet length (bidirectional) flow duration average data packet length average packet inter-arrival time
Fahad et al. [Fahad et al., 2014]	server port total number of packets with PUSH bit total number of bytes sent in the initial window (server to client) total number of RTT samples
Soysal and Schmidt [Soysal and Schmidt, 2010]	server port number client port number total number of packets total number of bytes flow duration service type flags type protocol type
Zhang et al. [Zhang et al., 2012]	server port minimum segment size (client to server) total number of bytes sent in the initial window (server to client) total number of bytes sent in the initial window (client to server)

5.2.2 Assessing the Machine Learning Classification Method

In this step, the collected TMs were randomly divided into two sets (with the same number of TMs), a training set used to feed the classifiers SVM and RF and a testing set used to classify the applications. The results reported in this section are the average of ten trials.

For comparing our classification results with the state-of-the-art, we extracted the feature vectors of four existing works that achieved significantly high classification accuracy: Eerman [Eerman et al., 2006], Fahad [Fahad et al., 2014], Soysal [Soysal and Schmidt, 2010], and Zhang [Zhang et al., 2012][9]. The information used for creating the feature vectors is presented in Table 5.1.

For every experiment, we have first tuned the classifiers as follows. We first tested different kernels for SVM and then, the kernel parameters γ and C were empirically defined through a grid search and fivefold cross-validation using the training set[10]. This operation was

[9]For simplicity, when referring to these works, we used only the first author's surname.

[10]`http://scikit-learn.org/stable/modules/svm.html`

also applied to tune the parameters for the RF classifier. All the experiments were carried out using scikit-learn [Pedregosa et al., 2011], an open-source ML library in Python.

Figure 5.6 shows the accuracy SVM and RF for classifying the applications. As can be seen, the overall accuracy obtained by the classifiers was higher when using features extracted from the TMs textures. RF achieved better results for all tested feature vectors, where the three highest values were 98.19%, 96.37%, and 87.65% for our approach (RLBP), our approach (ULBP), and Zhang [Zhang et al., 2012], respectively. When using the SVM classifier, our approach reached an accuracy of 94.87% with both LBP implementations and Zhang [Zhang et al., 2012] achieved 72.72%.

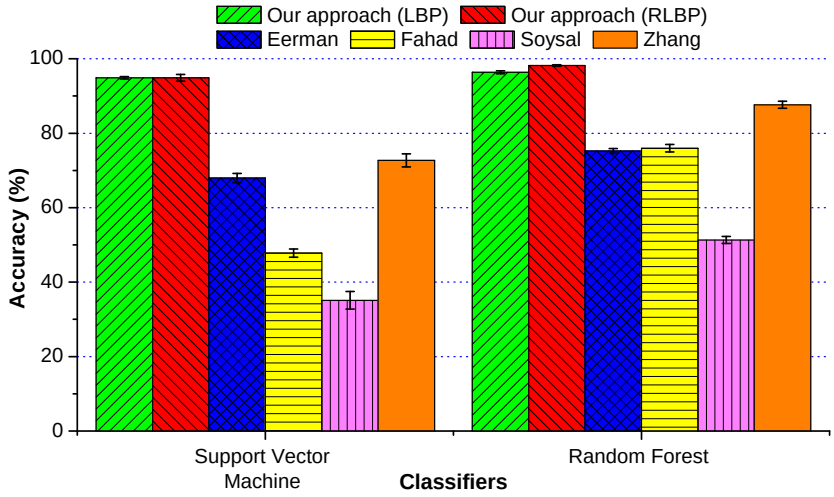


Figure 5.6: Accuracy of classifiers.

We observed in Figure 5.6, that the feature vectors described in the previously discussed literature (Eerman [Eerman et al., 2006], Fahad [Fahad et al., 2014], Zhang [Zhang et al., 2012], and especially Soysal [Soysal and Schmidt, 2010]) did not achieve the level of accuracy as reported in their respective papers. We believe that the contributing factor to this discrepancy was that all applications use the same server port (TCP port number 22) for encrypted communication and, as the server port is one feature used by them, it was not discriminative for classifying these applications. Another element that may contributed for this variance is the intrinsic communication randomness of the MapReduce shuffling phase.

Figure 5.7 illustrates the average classification time normalised to the maximum value, computed when SVM classified our approach (ULBP).. Our first observation is that RF was faster than SVM in all cases, this can be explained by the very different nature of the algorithms design. The results show that Eerman [Eerman et al., 2006], Fahad [Fahad et al., 2014], and Soysal [Soysal and Schmidt, 2010] achieved the highest classification speeds with the RF classifier, slightly higher than 20% of the highest measured time.

It is possible to observe a difference of 20% when SVM classified the two implementations of LBP. This difference occurred because, during the training phase, the SVM needed more support vectors for ULBP than for RLBP. With SVM the fastest time was for Eerman [Eerman et al., 2006], followed by Zhang, taking just under 40% of the maximum time. When comparing the different approaches with the RF classifier, we can see that our approach is about 10% slower than the other methods.

The time measured for RF to classify our approach is around 10% higher than that of other approaches. This difference in our testbed corresponds to 17.8 μs. On the other hand, the

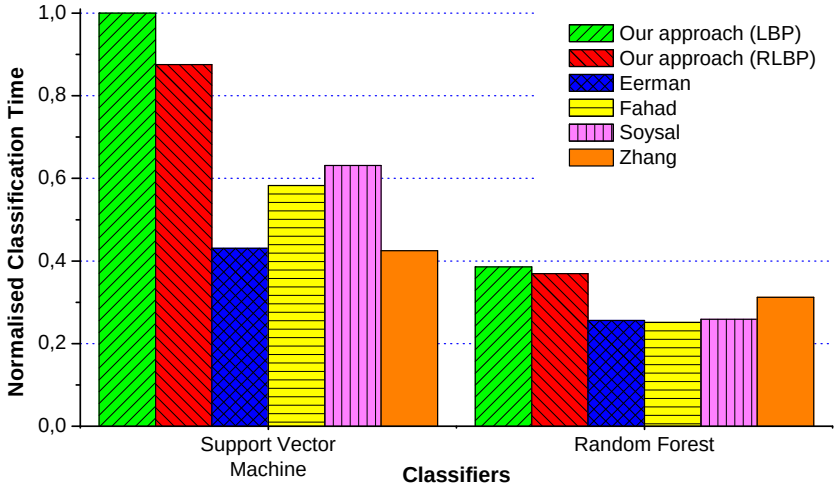


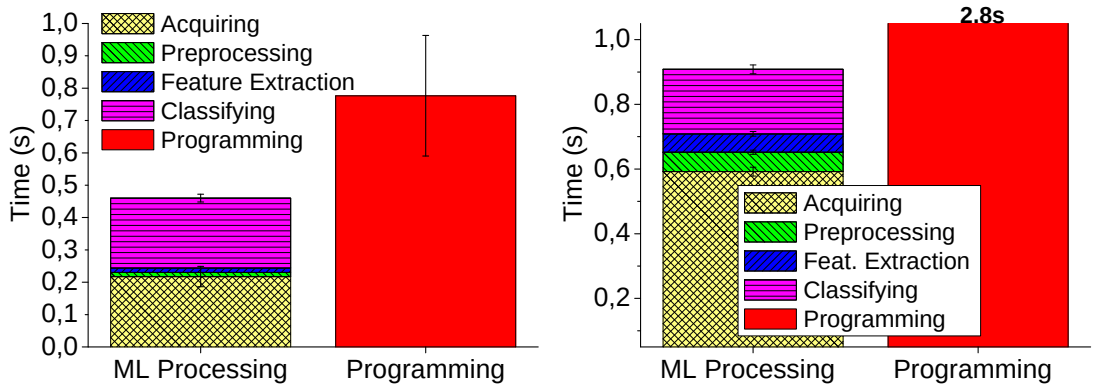
Figure 5.7: Normalized classification time.

accuracy of our method is less than 2% than the optimal solution, and the best accuracy obtained with the other methods was 10.5% worse than ours.

5.2.3 Overall Evaluation

In this second evaluation step, we executed four NPB tests (*bt*, *cg*, *lu*, and *mg*), setting up TReco to automatically recognize them. As the improvement in the execution time of *ft* benckmark with our approach was considerably higher than LACP, we decided not using it to avoid interfering the result. Before starting the evaluation, we ran TReco in the learning mode, collecting approximately 200 TMs from each benchmark, annotating them with the bandwidth and latency constraints, and storing them in TReco's database.

First, we evaluated the processing time for all stages of TReco in the running mode, considering 16 and 64 processes; these results are shown in Figure 5.8. We did not evaluate the times of learning mode because it is executed sporadically, only when new communication patterns have to be added.



(a) 16 processes.

(b) 64 processes.

Figure 5.8: Times measured for each TReco's processing stage.

The time for executing all ML processing, from acquiring to classifying a TM was, on average, 0.46 seconds for 16 processes and 0.9 seconds for 64 processes. For programming the

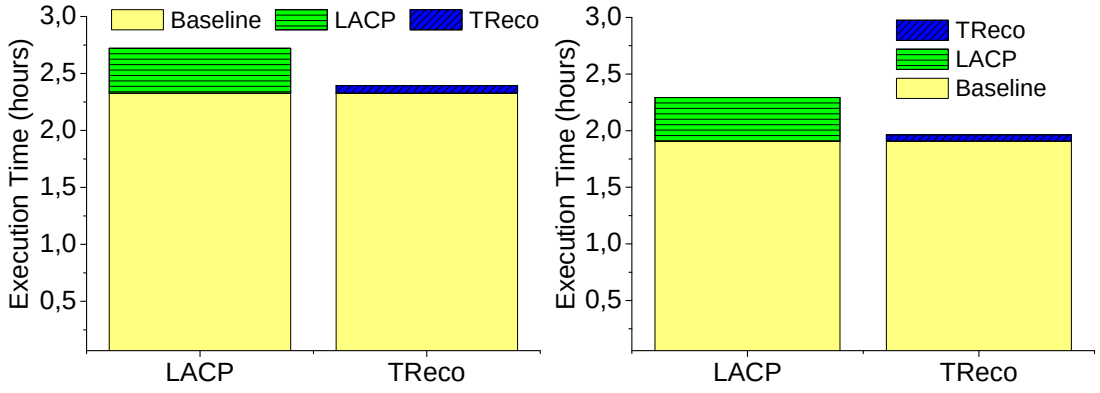
network devices, on average, we measured 0.77 seconds for 16 processes and 2.8 seconds for 64 processes; the standard deviation for this case, not shown in the figure, was 1.38 seconds.

The standard deviation for programming the network devices was considerable, this is explained by the substantial variation in the number of rules for each communication pattern. For example, to change the network forwarding, considering 64 processes, the total number of rules installed on the three switches were respectively 162, 48, 48, and 136 for the *bt*, *cg*, *lu*, and *mg* benchmarks.

Finally, to see how TReco would work in a real HPC environment, we used a Linux shell script to simulate a queuing system, adding to it several times, at random, a call to the benchmark tests known by the framework. For assessing it, we used the same methodology described in Section 4.2, connecting all computers to a single switch configured as L2/L3 mode to be the baseline. We also used the same topology described in Figure 4.11, measuring the times with our approach and with LACP.

We executed TReco in the running mode and started the queuing system. Our framework acquired, processed, and classified the TMs every four seconds, if it recognized three times consecutively the same communication pattern, it reprogrammed the switches according to that pattern, placing the latency-sensitive flows in latency paths and balancing the bandwidth-intensive communications across the bandwidth links.

The classification accuracy in this test was 98.5%, and we noticed that the most prediction errors occur when a benchmark finishes its execution and the next one starts executing. However, in our tests, the network was never reprogrammed erroneously, because TReco only reprograms it after three consecutive predictions.



(a) 16 processes. (b) 64 processes.

Figure 5.9: Random queue execution times.

Figure 5.9 shows the execution times for running the simulated queuing system. The time for executing all applications, considering all processes connected to the same switch (baseline), took 8379 seconds (2.32 hours) for 16 processes and 6865 seconds (1.9 hours) for 64 processes. Considering the overhead imposed by the topology, for 16 processes our approach increased the execution time in only 2.8%, while LACP was near to 17%. For 64 processes our approach increased the time in 4.4% and LACP worsened it in more than 20%. Once again we analyzed the statistics of the LACP ports, realized that it was not evenly balancing the flows.

We also analyzed the individual times of each benchmark, realizing, in average, an increase in the execution time of 8.2%, compared to the values obtained with the CPNP (reported in Section 4.2.3). This difference is explained by the time taken by the framework to "realize" that the network is running a different communication pattern, and also for reprogramming its devices.

5.3 Chapter Remarks

In this chapter we presented TReco, a system designed for automatically identifying the communication patterns and reprogramming the network behavior for speeding up the applications execution time. The system has two modes of operation, the first is used for learning the communication patterns and defining the bandwidth/latency constraints. The second, the running mode, is executed for automating the communication pattern identification and optimizing the network.

The chapter also reported a novel approach we proposed for classifying communication patterns using the textures of TMs as input for ML techniques. The approach consists of measuring the TMs and applying a preprocessing function for generating different visual textures for each pattern. The feature vectors are extracted through two methods widely used for texture recognition, the ULBP and the RLBP. These feature vectors are used as input for SVM and RF, two ML classifiers.

We performed experiments for classifying the traffic of four scientific applications plus two MapReduce execution phases, comparing the accuracy of our proposal with four methods described in current literature. As the applications use secure communication and the shuffling phase of MapReduce presents randomness in the communications among the computing nodes, the best accuracy measured for the other approaches was 87.6%. On the other hand, our method achieved approximately 98.2%, because it uses information extracted from the TMs and therefore has the advantage of having a global and comprehensive view of the network communications.

Texture is a powerful tool for classification, it has been used for classifying hundreds of classes in many different fields. For instance, Bertolini et al. has used the textures for identifying and verifying 650 different writers, achieving accuracies as higher as 99% [Bertolini et al., 2013]. We thus concluded that it is possible to use different information from that commonly used (i.e. intrinsic and statistical flow information) for traffic classification, and extracting the textures from TMs proved to be a good option.

Another important consideration is the time for executing the entire process. It took, in average, less than one second from acquiring the TMs to classify the applications. For programming the network devices, the average measured time was 2.8 seconds. If we consider that the execution time of a MapReduce or a scientific application may take several hours, we perceive that the time used by our approach is negligible and that our approach can be used.

Finally, for evaluating a queuing system, we executed, in random order, communication patterns known by TReco, using the framework to automatically recognize these patterns and to program the network for optimizing them. We measured the execution time for the entire queue when the communications were balanced with the LACP and with our approach, realizing a decrease in time of 15.8% when the network was programmed by TReco.

Chapter 6

Conclusion

This work aimed to investigate how the communication patterns expressed by HPC applications could be used in conjunction with SDN to create abstractions that would allow optimizing the network according to the communications requirements, with the final purpose of reducing the execution time of these applications.

For understanding the abstractions offered by SDN, we explored its different programming levels, surveying the feature of all SDN programming languages, searching for abstractions that could help us achieve our goal. This study revealed a wide range of features with considerable amount of intersections among them. Due to the absence of standardization and the lack of consensus on the terminology used for naming these abstractions, we proposed a taxonomy for grouping them. We identified nineteen different features, divided into ten categories. To reach the objective proposed in this thesis, we used windowed history monitoring abstraction for collecting the TMs and we applied a path selection abstraction for placing the application flows on the network paths.

We studied many SDN related work aiming to speed up applications, realizing that the most used techniques are changing the network forwarding or applying QoS; higher-level abstractions are often offered to facilitate these operations. We also study the issues introduced by the SDN programmability and we observed that the main problems are the time for querying the controller to populate the network devices' flow tables and, due to the huge number of matching rules installed in these tables, the time to perform a lookup on them.

For better understanding the HPC applications' communication patterns, we analyzed their TMs, inspecting the spatial and temporal behaviors. We concluded that HPC applications tend to transmit the same amount of data across the same computing nodes, independently of the input data or the number of processing nodes; we named this as "well-behaved communication patterns." We realized that the TMs could be used for identifying the different communication patterns and, as the communications are well-behaved, the TMs also could be used for identifying the communications' requirements.

So, we developed a framework for placing the HPC applications communications through the network available paths. Our approach relied on storing the application TMs and annotating them with bandwidth and latency constraints. The framework provides an API that enable the HPC application developer to specify which communication pattern his application uses and, based on the previous stored TMs, the bandwidth-intensive and latency-sensitive communications are evenly placed on the network paths. The framework avoids the issues introduced by SDN by proactively installing the necessary flows and reduce the number of flow entries by grouping the flows by destination address.

We evaluated the framework analyzing the impact of network programmability, the consequences of multiple flows sharing the same path, and the delay caused by path length on point-to-point and collective communications. For our experiments, we used a real testbed, comparing our solution with LACP, a well-known protocol for balancing the flows over multiple network links. We used HPC benchmarks and real applications for assessing our proposal; at best, it halved the execution time of benchmarks and reduced the time of real applications up to 26.5%.

The proposed framework abstracts the complex, tedious, and error-prone task of programming the network, but requires the application to be modified including calls to its API. So, we developed an automatic solution for detecting the communication patterns. As HPC applications may adopt cryptographic methods for ensuring security in the communication, we decided using a ML approach. As we were already acquiring the TMs, we developed a new method using them. The problem is that the queue manager (or task scheduler) can return the computational nodes in random order so, we developed a function to unscramble the TMs. This function generates similar visual textures for each communication pattern, regardless the order the nodes were allocated. Two well-known textural representations (ULBP and RLBP) were used for feeding the ML classifiers. Our methodology for detecting the communication patterns succeeded with an accuracy rate over 98% in our experiments.

Finally, the ML method was incorporated into the framework, generating an autonomic solution that abstracts the network programming by automatically identifying the communication pattern and modifying the network forwarding. For evaluating the framework, we created a queue, , in random order, of different communication patterns. We computed the total time for executing this queue when the network was configured with LACP and we also ran our framework for automatically identifying these communication patterns and modifying the network behavior. We observed a reduction in the execution time of 15.8% when the network was programmed with our approach.

This work proves that the execution time of the HPC applications can be reduced if their communication patterns are used as input for reprogramming the SDN-enabled devices, tuning the network according to the requirements of each communication pattern. However, we use only a subset of the existing techniques to demonstrate our initial thesis, leaving many opportunities for future research.

As stated before, we used windowed history monitoring and path selection for tuning the network. One possible extension of our work is investigating how the other SDN abstractions can be used for improving the performance of HPC applications. One possibility is to apply QoS rules on the latency-sensitive communications for prioritizing them. We also do not consider network failures then fault tolerance mechanisms can be incorporated into our work to make it fault-tolerant. Also, as future work, security issues can be studied and addressed.

Our proposal considers the applications running in a dedicated cluster so, it can be extended to an environment with several applications executing simultaneously. In this case, virtual network slices can be created, one for each application, and our method can be used for tuning each network slice. A coordinator module will also be needed for handling the intersections of the flows allocated in the different slices.

Lastly, some aspects that our prototype still requires user intervention may be automated. For example, read the topology from controller and automatically set the bandwidth-intensive and latency-sensitive paths, employ heuristics to analyze TMs and automatically identify latency and bandwidth constraints, and improve the method to classify the communication patterns by employing ML techniques to automatically identify these patterns.

Publications

Below we list the publications from this work.

Trois, C., Martinello, M., Bona, L., and Del Fabro, M. (2015). From Software Defined Network To Network Defined for Software. In *Proceedings of the 2015 ACM Symposium on Applied Computing*. ACM.

Trois, C., de Bona, L. C. E., Fabro, M. D. D. D., and Martinello, M. (2016). Carving Software-Defined Networks for Scientific Applications with SpateN. In 41st *Conference on Local Computer Networks (LCN)*, pages 606–610. IEEE.

Trois, C., Fabro, M. D. D., de Bona, L. C. E., and Martinello, M. (2016). A survey on sdn programming languages: Towards a taxonomy. *IEEE Communications Surveys Tutorials*, 18(4):2687–2712.

Trois, C., de Bona, L. C. E., Fabro, M. D. D., Martinello, M., Bidkar, S., Nejabati, R., and Simeonidou, D. (2017). Softening up the network for scientific applications. In *25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 410–418. IEEE.

Trois, C., Weingaertner, D., Pasqualin, D., Maciel, E., Almeida, E., Silva, F., Tissot, H., de Bona, L. C. E., Castilho, M., Fabro, M. D. D., Sunyé, M. (2017). Transparency Meets Management: a Monitoring and Evaluating Tool for Governmental Projects. In *ACS International Conference on Computer Systems and Applications (AICCSA)*, IEEE.

Bibliography

[Achour and Nasri, 2012] Achour, S. and Nasri, W. (2012). A performance prediction approach for mpi routines on multi-clusters. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 125–129. IEEE.

[Ahern et al., 2007] Ahern, S., Alam, S. R., Fahey, M. R., Hartman-Baker, R. J., Barrett, R. F., Kendall, R. A., Kothe, D. B., Mills, R. T., Sankaran, R., Tharrington, A. N., et al. (2007). Scientific application requirements for leadership computing at the exascale. Technical report, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences.

[Ahmed et al., 2015] Ahmed, M. F., Talhi, C., and Cheriet, M. (2015). Towards flexible, scalable and autonomic virtual tenant slices. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 720–726. IEEE.

[Akyildiz et al., 2014] Akyildiz, I. F., Lee, A., Wang, P., Luo, M., and Chou, W. (2014). A roadmap for traffic engineering in SDN-OpenFlow networks. *Computer Networks*, 71:1–30.

[Al-Fares et al., 2010] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. (2010). Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19.

[Al-Shabibi et al., 2014] Al-Shabibi, A., De Leenheer, M., Gerola, M., Koshibe, A., Snow, W., and Parulkar, G. (2014). Openvirtex: A network hypervisor. *Open Networking Summit*.

[Alsmadi et al., 2016] Alsmadi, I., Khamaiseh, S., and Xu, D. (2016). Network parallelization in hpc clusters. In *Computational Science and Computational Intelligence (CSCI), 2016 International Conference on*, pages 584–589. IEEE.

[Amsden, 2011] Amsden, E. (2011). A survey of functional reactive programming. *Unpublished*.

[Anderson et al., 2014] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. (2014). NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126.

[Armstrong, 2013] Armstrong, J. (2013). *Programming Erlang*. Pragmatic Bookshelf.

[Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., et al. (2006). The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[Asanovic et al., 2008] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J. D., Lee, E. A., Morgan, N., Necula, G., Patterson, D. A., et al. (2008). The parallel computing laboratory at uc berkeley: A research agenda based on the berkeley view. *EECS Department, University of California, Berkeley, Tech. Rep*.

[Ash, 2001] Ash, G. R. (2001). Traffic Engineering & QoS Methods for IP, ATM, & TDM-Based Multi-service Networks. In *Internet-Draft, Work in Progress*.

[Autenrieth et al., 2013] Autenrieth, A., Elbers, J.-P., Kaczmarek, P., and Kostecki, P. (2013). Cloud orchestration with sdn/openflow in carrier transport networks. In *Transparent Optical Networks (ICTON), 2013 15th International Conference on*, pages 1–4. IEEE.

[Awduche et al., 2002] Awduche, D., Chiu, A., Elwalid, A., Widjaja, I., and Xiao, X. (2002). Overview and Principles of Internet Traffic Engineering. RFC 3272, RFC Editor.

[Awduche et al., 1999] Awduche, D., Malcolm, J., Agogbua, J., O'Dell, M., and McManus, J. (1999). Requirements for Traffic Engineering Over MPLS. RFC 2702, RFC Editor.

[Bailey et al., 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

[Bainomugisha et al., 2012] Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S., and De Meuter, W. (2012). A survey on reactive programming. In *ACM Computing Surveys*.

[Banikazemi et al., 2013] Banikazemi, M., Olshefski, D., Shaikh, A., Tracey, J., and Wang, G. (2013). Meridian: an sdn platform for cloud network services. *Communications Magazine, IEEE*, 51(2):120–127.

[Barkai, 2009] Barkai, D. (2009). The application perspective: Seeking productivity and performance. *The International Journal of High Performance Computing Applications*, 23(4):403–408.

[Barrow-Williams et al., 2009] Barrow-Williams, N., Fensch, C., and Moore, S. (2009). A communication characterisation of splash-2 and parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 86–97. IEEE.

[Batalle et al., 2013] Batalle, J., Ferrer Riera, J., Escalona, E., and Garcia-Espin, J. A. (2013). On the implementation of NFV over an OpenFlow infrastructure: Routing Function Virtualization. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pages 1–6. IEEE.

[Batista and Fernandez, 2014] Batista, B. L. A. and Fernandez, M. P. (2014). PonderFlow: A Policy Specification Language for Openflow Networks. *ICN 2014*, page 215.

[Batory, 2005] Batory, D. (2005). *Feature models, grammars, and propositional formulas*. Springer.

[Beach, 2010] Beach, T. H. O. (2010). *Application Acceleration: An Investigation of Automatic Porting*. PhD thesis, Cardiff University.

[Benson et al., 2010] Benson, T., Akella, A., and Maltz, D. A. (2010). Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM.

[Bernaille et al., 2006] Bernaille, L., Teixeira, R., Akodkenou, I., Soule, A., and Salamatian, K. (2006). Traffic classification on the fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26.

[Bertolini et al., 2013] Bertolini, D., Oliveira, L. S., Justino, E., and Sabourin, R. (2013). Texture-based descriptors for writer identification and verification. *Expert Systems with Applications*, 40(6):2069–2080.

[Bosshart et al., 2014] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

[Botelho et al., 2014] Botelho, F., Bessani, A., Ramos, F., and Ferreira, P. (2014). On the design of practical fault-tolerant SDN controllers. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 73–78. IEEE.

[Bouet et al., 2013] Bouet, M., Leguay, J., and Conan, V. (2013). Cost-based placement of virtualized deep packet inspection functions in sdn. In *Military Communications Conference, MILCOM 2013-2013 IEEE*, pages 992–997. IEEE.

[Bozakov and Papadimitriou, 2012] Bozakov, Z. and Papadimitriou, P. (2012). Autoslice: automated and scalable slicing for software-defined networks. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 3–4. ACM.

[Braga et al., 2010] Braga, R., Mota, E., and Passito, A. (2010). Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 408–415. IEEE.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[Briggs et al., 2014] Briggs, S., Karney, B. W., and Sleep, B. E. (2014). Numerical modelling of flow and transport in rough fractures. *Journal of Rock Mechanics and Geotechnical Engineering*, 6(6):535–545.

[Bujlow et al., 2013] Bujlow, T., Carela-Español, V., and Barlet-Ros, P. (2013). Comparison of deep packet inspection (dpi) tools for traffic classification. Technical report, Universitat Politècnica de Catalunya.

[Casado et al., 2014] Casado, M., Foster, N., and Guha, A. (2014). Abstractions for software-defined networks. *Communications of the ACM*, 57(10):86–95.

[Casado et al., 2010] Casado, M., Koponen, T., Ramanathan, R., and Shenker, S. (2010). Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 8. ACM.

[Chandrasekaran and Benson, 2014] Chandrasekaran, B. and Benson, T. (2014). Tolerating SDN application failures with LegoSDN. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 22. ACM.

[Che et al., 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE.

[Chodnekar et al., 1997] Chodnekar, S., Srinivasan, V., Vaidya, A. S., Sivasubramaniam, A., and Das, C. R. (1997). Towards a communication characterization methodology for parallel applications. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 310–319. IEEE.

[Chowdhury and Boutaba, 2010] Chowdhury, N. M. K. and Boutaba, R. (2010). A survey of network virtualization. *Computer Networks*, 54(5):862–876.

[Chowdhury et al., 2014] Chowdhury, S. R., Bari, M. F., Ahmed, R., and Boutaba, R. (2014). Payless: A low cost network monitoring framework for software defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9. IEEE.

[Clocksin and Mellish, 2003] Clocksin, W. and Mellish, C. S. (2003). *Programming in PROLOG*. Springer Science & Business Media.

[Cofano et al., 2014] Cofano, G., De Cicco, L., and Mascolo, S. (2014). A control architecture for massive adaptive video streaming delivery. In *Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming*, pages 7–12. ACM.

[Colella, 2004] Colella, P. (2004). Defining software requirements for scientific computing.

[Courtney et al., 2003] Courtney, A., Nilsson, H., and Peterson, J. (2003). The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM.

[CPqD, 2017] CPqD, P. (2017). OpenFlow 1.3 Software Switch by CPqD. Website. `http://cpqd.github.io/ofsoftswitch13/`. Accessed 11 May 2017.

[Curtis et al., 2011] Curtis, A. R., Kim, W., and Yalagandula, P. (2011). Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE*, pages 1629–1637. IEEE.

[Cypher et al., 1993] Cypher, R., Ho, A., Konstantinidou, S., and Messina, P. (1993). *Architectural requirements of parallel scientific applications with explicit communication*, volume 21. ACM.

[Czaplicki and Chong, 2013] Czaplicki, E. and Chong, S. (2013). Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM.

[Das et al., 2013] Das, A., Lumezanu, C., Zhang, Y., Singh, V., Jiang, G., and Yu, C. (2013). Transparent and flexible network management for big data processing in the cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*.

[Das et al., 2011] Das, S., Yiakoumis, Y., Parulkar, G., McKeown, N., Singh, P., Getachew, D., and Desai, P. D. (2011). Application-aware aggregation and traffic engineering in a converged packet-circuit network. In *National Fiber Optic Engineers Conference*, page NThD3. Optical Society of America.

[Date et al., 2015] Date, S., Abe, H., Khureltulga, D., Takahashi, K., Kido, Y., Watashiba, Y., Pongsakorn, U., Ichikawa, K., Yamanaka, H., Kawai, E., et al. (2015). An empirical study of sdn-accelerated hpc infrastructure for scientific research. In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pages 89–96. IEEE.

[Dhote et al., 2015] Dhote, Y., Agrawal, S., and Deen, A. J. (2015). A survey on feature selection techniques for internet traffic classification. In *Computational Intelligence and Communication Networks (CICN), 2015 International Conference on*, pages 1375–1380. IEEE.

[Diener et al., 2015] Diener, M., Cruz, E. H., Pilla, L. L., Dupros, F., and Navaux, P. O. (2015). Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation*, 88:18–36.

[Dimond et al., 2011] Dimond, R., Racaniere, S., and Pell, O. (2011). Accelerating large-scale hpc applications using fpgas. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 191–192. IEEE.

[Doria et al., 2010] Doria, A., Salim, J. H., Haas, R., Khosravi, H., Wang, W., Dong, L., Gopal, R., and Halpern, J. (2010). Forwarding and control element separation (forces) protocol specification. *Internet Requests for Comments, RFC Editor, RFC*, 5810.

[Drutskoy et al., 2013] Drutskoy, D., Keller, E., and Rexford, J. (2013). Scalable network virtualization in software-defined networks. *Internet Computing, IEEE*, 17(2):20–27.

[Eerman et al., 2006] Eerman, J., Mahanti, A., and Arlitt, M. (2006). Internet traffic identification using machine learning techniques. In *Proc. of the 49th IEEE Global Telecomm. Conf.(GLOBECOM)*, pages 1–6.

[Egilmez et al., 2013] Egilmez, H. E., Civanlar, S., et al. (2013). An optimization framework for QoS-enabled adaptive video streaming over OpenFlow networks. *Multimedia, IEEE Transactions on*, 15(3):710–715.

[Egilmez et al., 2012] Egilmez, H. E., Dane, S. T., Bagci, K. T., et al. (2012). OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks. In *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*, pages 1–8. IEEE.

[Erickson, 2013] Erickson, D. (2013). The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM.

[Erlacher et al.,] Erlacher, C., Jankowski, P., Blaschke, T., Paulus, G., and Anders, K.-H. A gpu-based parallelization approach to conduct spatially-explicit uncertainty and sensitivity analysis in the application domain of landscape assessment. *GI_Forum 2017,*, 1:44–58.

[Fahad et al., 2014] Fahad, A., Tari, Z., Khalil, I., Almalawi, A., and Zomaya, A. Y. (2014). An optimal and stable feature selection approach for traffic classification based on multi-criterion fusion. *Future Generation Computer Systems*, 36:156–169.

[Fan et al., 2011] Fan, W., Li, J., Ma, S., Tang, N., and Wu, Y. (2011). Adding regular expressions to graph reachability and pattern queries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 39–50. IEEE.

[Feamster and Balakrishnan, 2005] Feamster, N. and Balakrishnan, H. (2005). Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56. USENIX Association.

[Feamster et al., 2013] Feamster, N., Rexford, J., and Zegura, E. (2013). The road to SDN. *Queue*, 11(12):20.

[Feitelson and Weil, 1998] Feitelson, D. G. and Weil, A. M. (1998). Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 542–546. IEEE.

[Feng et al., 2012] Feng, W.-c., Lin, H., Scogland, T., and Zhang, J. (2012). Opencl and the 13 dwarfs: a work in progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 291–294. ACM.

[Feng et al., 2009] Feng, Y., Guo, R., Wang, D., and Zhang, B. (2009). Research on the Active DDoS Filtering Algorithm Based on IP Flow. In *Natural Computation, 2009. ICNC'09. Fifth International Conference on*, volume 4, pages 628–632. IEEE.

[Ferg, 2006] Ferg, S. (2006). *Event-driven programming: Introduction, tutorial, history*. Autoedicion.

[Ferguson et al., 2012a] Ferguson, A., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. (2012a). Hierarchical policies for software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 37–42. ACM.

[Ferguson et al., 2013] Ferguson, A. D., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. (2013). Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 327–338. ACM.

[Ferguson et al., 2012b] Ferguson, A. D., Guha, A., Place, J., Fonseca, R., and Krishnamurthi, S. (2012b). Participatory networking. *Proc. Hot-ICE*, 12.

[Fernandes and Rothenberg, 2014] Fernandes, E. L. and Rothenberg, C. E. (2014). Openflow 1.3 software switch. *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC, pgs*, pages 1021–1028.

[Fernandez, 2013] Fernandez, M. P. (2013). Comparing openflow controller paradigms scalability: Reactive and proactive. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 1009–1016. IEEE.

[Floodlight, 2017] Floodlight, P. (2017). Indigo Virtual Switch. Website. `http://www.projectfloodlight.org/indigo-virtual-switch`. Accessed 11 May 2017.

[Foster et al., 2010] Foster, N., Freedman, M. J., Harrison, R., Rexford, J., Meola, M. L., and Walker, D. (2010). Frenetic: a high-level language for OpenFlow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 6. ACM.

[Foster et al., 2013] Foster, N., Guha, A., Reitblatt, M., Story, A., Freedman, M. J., Katta, N. P., Monsanto, C., Reich, J., Rexford, J., Schlesinger, C., et al. (2013). Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134.

[Foster et al., 2011] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. (2011). Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM.

[Galante et al., 2016] Galante, G., De Bona, L. C. E., Mury, A. R., Schulze, B., and da Rosa Righi, R. (2016). An analysis of public clouds elasticity in the execution of scientific applications: a survey. *Journal of Grid Computing*, 14(2):193–216.

[Gember et al., 2012] Gember, A., Prabhu, P., Ghadiyali, Z., and Akella, A. (2012). Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 7–12. ACM.

[Gémieux et al., 2017] Gémieux, M., Savaria, Y., David, J.-P., and Zhu, G. (2017). A cache-coherent heterogeneous architecture for low latency real time applications. In *Real-Time Distributed Computing (ISORC), 2017 IEEE 20th International Symposium on*, pages 176–184. IEEE.

[Ghorbani and Caesar, 2012] Ghorbani, S. and Caesar, M. (2012). Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 67–72. ACM.

[Giotis et al., 2014] Giotis, K., Androulidakis, G., and Maglaris, V. (2014). Leveraging SDN for efficient anomaly detection and mitigation on legacy networks. In *Proceedings of the third European Workshop on Software Defined Networks (EWSDN)*, pages 85–90. IEEE.

[Gong et al., 2015] Gong, Y., Wang, X., Malboubi, M., Wang, S., Xu, S., and Chuah, C.-N. (2015). Towards accurate online traffic matrix estimation in software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 26. ACM.

[Goransson and Black, 2014] Goransson, P. and Black, C. (2014). *Software Defined Networks: A Comprehensive Approach*. Elsevier.

[Gude et al., 2008] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. (2008). Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110.

[Gupta and Milojicic, 2011] Gupta, A. and Milojicic, D. (2011). Evaluation of hpc applications on cloud. In *Open Cirrus Summit (OCS), 2011 Sixth*, pages 22–26. IEEE.

[Gutz et al., 2012] Gutz, S., Story, A., Schlesinger, C., and Foster, N. (2012). Splendid isolation: A slice abstraction for software-defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 79–84. ACM.

[Hakiri and Gokhale, 2016] Hakiri, A. and Gokhale, A. (2016). Data-centric publish/subscribe routing middleware for realizing proactive overlay software-defined networking. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 246–257. ACM.

[Halloway, 2009] Halloway, S. (2009). *Programming Clojure*. Pragmatic Bookshelf.

[He and Shenoy, 2016] He, X. and Shenoy, P. (2016). Firebird: Network-aware task scheduling for spark using sdns. In *Computer Communication and Networks (ICCCN), 2016 25th International Conference on*, pages 1–10. IEEE.

[Heuveline and Latt, 2007] Heuveline, V. and Latt, J. (2007). The openlb project: an open source and object oriented implementation of lattice boltzmann methods. *International Journal of Modern Physics C*, 18(04):627–634.

[Hinrichs et al., 2009] Hinrichs, T. L., Gude, N. S., Casado, M., Mitchell, J. C., and Shenker, S. (2009). Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 1–10. ACM.

[Jain, 2016] Jain, N. (2016). *Optimization of communication intensive applications on HPC networks*. PhD thesis, University of Illinois at Urbana-Champaign.

[Jain and Paul, 2013] Jain, R. and Paul, S. (2013). Network virtualization and software defined networking for cloud computing: a survey. *Communications Magazine, IEEE*, 51(11):24–31.

[Jain et al., 2013] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al. (2013). B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM.

[Jakl, 2008] Jakl, M. (2008). Rest representational state transfer.

[Jamalian and Rajaei, 2015] Jamalian, S. and Rajaei, H. (2015). Data-intensive hpc tasks scheduling with sdn to enable hpc-as-a-service. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 596–603. IEEE.

[Jammal et al., 2014] Jammal, M., Singh, T., Shami, A., Asal, R., and Li, Y. (2014). Software defined networking: State of the art and research challenges. *Computer Networks*, 72:74–98.

[Jarschel et al., 2013] Jarschel, M., Wamser, F., Hohn, T., Zinner, T., and Tran-Gia, P. (2013). Sdn-based application-aware networking on the example of youtube video streaming. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 87–92. IEEE.

[Julkunen and Chow, 1998] Julkunen, H. and Chow, C. E. (1998). Enhance network security with dynamic packet filter. In *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*, pages 268–275. IEEE.

[Kakadia et al., 2013] Kakadia, D., Kopri, N., and Varma, V. (2013). Network-aware virtual machine consolidation for large data centers. In *Proceedings of the Third International Workshop on Network-Aware Data Management*, page 6. ACM.

[Kang et al., 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document.

[Kang et al., 2013] Kang, N., Liu, Z., Rexford, J., and Walker, D. (2013). Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 13–24. ACM.

[Kannan and Banerjee, 2013] Kannan, K. and Banerjee, S. (2013). Compact tcam: Flow entry compaction in tcam for power aware sdn. In *Distributed Computing and Networking*, pages 439–444. Springer.

[Katta et al., 2012] Katta, N. P., Rexford, J., and Walker, D. (2012). Logic programming for software-defined networks. In *Workshop on Cross-Model Design and Validation (XLDI)*, volume 412.

[Katta et al., 2013] Katta, N. P., Rexford, J., and Walker, D. (2013). Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM.

[Kawai, 2012] Kawai, E. (2012). Can sdn help hpc? In *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on*, pages 210–210. IEEE.

[Keller et al., 2012] Keller, E., Ghorbani, S., Caesar, M., and Rexford, J. (2012). Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 109–114. ACM.

[Kim et al., 2015] Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., and Clark, R. (2015). Kinetic: Verifiable Dynamic Network Control. In *USENIX NSDI*.

[Kim et al., 2012] Kim, H., Santos, J. R., Turner, Y., Schlansker, M., Tourrilhes, J., and Feamster, N. (2012). Coronet: Fault tolerance for software defined networks. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–2. IEEE.

[Kim and Lilja, 1998] Kim, J. and Lilja, D. J. (1998). *Characterization of communication patterns in message-passing parallel scientific application programs*. Springer.

[Klaedtke et al., 2014] Klaedtke, F., Karame, G. O., Bifulco, R., and Cui, H. (2014). Access control for SDN controllers. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 219–220. ACM.

[Koponen et al., 2014] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Gude, N., Ingram, P., et al. (2014). Network virtualization in multi-tenant datacenters. In *USENIX NSDI*.

[Koponen et al., 2010] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., et al. (2010). Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6.

[Kotronis et al., 2014] Kotronis, V., Dimitropoulos, X., Klöti, R., Ager, B., Georgopoulos, P., and Schmid, S. (2014). Control exchange points: Providing qos-enabled end-to-end services via sdn-based inter-domain routing orchestration. *LINX*, 2429(1093):2443.

[Kreutz et al., 2015] Kreutz, D., Ramos, F. M., Verissimo, P., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76.

[Kumar et al., 2015] Kumar, A., Jain, S., Naik, U., Raghuraman, A., Carlin, B., Amarandei-Stavila, M., Robin, M., Siganporia, A., Stuart, S., and Vahdat, A. (2015). Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 1–14. ACM.

[Kuster, 2017] Kuster, L. (2017). *dCUDA: GPU Cluster Programming using IB Verbs*. PhD thesis, Department of Computer Science, ETH Zurich.

[Lang, 2005] Lang, J. (2005). Link Management Protocol (LMP). RFC 4204, RFC Editor.

[Lantz et al., 2010] Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM.

[Lee et al., 2014] Lee, J., Turner, Y., Lee, M., Popa, L., Banerjee, S., Kang, J.-M., and Sharma, P. (2014). Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 467–478. ACM.

[Li and Mitchell, 2003] Li, N. and Mitchell, J. C. (2003). Datalog with constraints: A foundation for trust management languages. In *Practical Aspects of Declarative Languages*, pages 58–73. Springer.

[Lifschitz, 2008] Lifschitz, V. (2008). What Is Answer Set Programming? In *AAAI*, volume 8, pages 1594–1597.

[Linc, 2017] Linc, s. (2017). Linc-switch. Website. `http://flowforwarding.github.io/LINC-Switch/`. Accessed 11 May 2017.

[Linux, 2017] Linux, F. (2017). Open vSwitch. Website. `http://openvswitch.org/`. Accessed 11 May 2017.

[Liu et al., 2015] Liu, J., Zhu, L., Sun, W., and Hu, W. (2015). Scalable application-aware resource management in software defined networking. In *Transparent Optical Networks (ICTON), 2015 17th International Conference on*, pages 1–5. IEEE.

[Liu and Xu, 2012] Liu, K. and Xu, K. (2012). OAuth based authentication and authorization in open telco API. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, volume 1, pages 176–179. IEEE.

[Lloyd, 2012] Lloyd, J. W. (2012). *Foundations of logic programming*. Springer Science & Business Media.

[Long et al., 2013] Long, H., Shen, Y., Guo, M., and Tang, F. (2013). LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 290–297. IEEE.

[Mahajan and Wattenhofer, 2013] Mahajan, R. and Wattenhofer, R. (2013). On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 20. ACM.

[Martinello et al., 2014] Martinello, M., Ribeiro, M., de Oliveira, R., and de Angelis Vitoi, R. (2014). Keyflow: a prototype for evolving SDN toward core network fabrics. *Network, IEEE*, 28(2):12–19.

[Matias et al., 2014] Matias, J., Garay, J., Mendiola, A., Toledo, N., and Jacob, E. (2014). FlowNAC: Flow-based network access control. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 79–84. IEEE.

[McCauley, 2015] McCauley, M. (2015). About POX.

[McKeown et al., 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

[Medved et al., 2014] Medved, J., Varga, R., Tkacik, A., and Gray, K. (2014). Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE.

[Mernik et al., 2005] Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344.

[Meyerovich et al., 2009] Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). Flapjax: a programming language for Ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM.

[Mitchell, 1996] Mitchell, J. C. (1996). *Foundations for programming languages*, volume 1. MIT press Cambridge.

[Monsanto et al., 2012] Monsanto, C., Foster, N., Harrison, R., and Walker, D. (2012). A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, 47(1):217–230.

[Monsanto et al., 2013] Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D., et al. (2013). Composing Software Defined Networks. In *NSDI*, pages 1–13.

[Mottola, 2005] Mottola, A. (2005). *Design and implementation of a declarative programming language in a reactive environment*. PhD thesis, Università degli Studi di Roma.

[Nadeau and Gray, 2013] Nadeau, T. D. and Gray, K. (2013). *SDN: software defined networks*. " O'Reilly Media, Inc.".

[Nam et al., 2014] Nam, H., Kim, K.-H., Kim, J. Y., and Schulzrinne, H. (2014). Towards qoe-aware video streaming using sdn. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 1317–1322. IEEE.

[Narayan et al., 2012] Narayan, S., Bailey, S., Greenway, M., Grossman, R., Heath, A., Powell, R., and Daga, A. (2012). Openflow enabled hadoop over local and wide area clusters. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1625–1628. IEEE.

[Narayana et al., 2014] Narayana, S., Rexford, J., and Walker, D. (2014). Compiling path queries in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 181–186. ACM.

[Nascimento et al., 2011] Nascimento, M. R., Rothenberg, C. E., Salvador, M. R., Corrêa, C. N., de Lucena, S. C., and Magalhães, M. F. (2011). Virtual routers as a service: the routeflow approach leveraging software-defined networks. In *Proceedings of the 6th International Conference on Future Internet Technologies*, pages 34–37. ACM.

[Nelson et al., 2014] Nelson, T., Ferguson, A. D., Scheer, M. J., and Krishnamurthi, S. (2014). Tierless programming and reasoning for software-defined networks. *NSDI, Apr*.

[Nelson et al., 2013] Nelson, T., Guha, A., Dougherty, D. J., Fisler, K., and Krishnamurthi, S. (2013). A balance of power: Expressive, analyzable controller programming. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 79–84. ACM.

[Networks, 2015] Networks, B. S. (2015). Project floodlight. Website. `http://www.projectfloodlight.org/`. Accessed 10 apr. 2015.

[Noyes et al., 2014] Noyes, A., Warszawski, T., Černỳ, P., and Foster, N. (2014). Toward synthesis of network updates. *arXiv preprint arXiv:1403.7840*.

[Nugteren, 2017] Nugteren, C. (2017). Clblast: A tuned opencl blas library. *arXiv preprint arXiv:1705.05249*.

[Ojala et al., 2002] Ojala, T., Pietikainen, M., and Maenpaa, T. (2002). Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Transactions on pattern analysis and machine intelligence*, 24(7):971–987.

[ONF, 2014] ONF (2014). Openflow switch specification version 1.5.0.

[ONF, 2017] ONF, N. (2017). Sdn nbi community. Website. `https://www.opennetworking.org/technical-communities/areas/services/1916-northbound-interfaces/`. Accessed 12 may 2017.

[ONF, 2013] ONF, O. (2013). Sdn architecture overview. Website. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf`. Accessed 12 mar. 2017.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Pfaff and Davie, 2013] Pfaff, B. and Davie, B. (2013). The open vswitch database management protocol.

[Pfaff et al., 2009] Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., and Shenker, S. (2009). Extending Networking into the Virtualization Layer. In *Hotnets*.

[Polezhaev et al., 2014] Polezhaev, P., Shukhman, A., and Ushakov, Y. (2014). Network resource control system for hpc based on sdn. In *International Conference on Next Generation Wired/Wireless Networking*, pages 219–230. Springer.

[Prabhakar et al., 2017] Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C., and Olukotun, K. (2017). Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 389–402. ACM.

[Qadir and Hasan, 2015] Qadir, J. and Hasan, O. (2015). Applying Formal Methods to Networking: Theory, Techniques, and Applications. *Communications Surveys & Tutorials, IEEE*, 17(1):256–291.

[Qazi et al., 2013] Qazi, Z. A., Lee, J., Jin, T., Bellala, G., Arndt, M., and Noubir, G. (2013). Application-awareness in sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 487–488. ACM.

[Qin et al., 2014] Qin, P., Dai, B., Huang, B., and Xu, G. (2014). Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data. *arXiv preprint arXiv:1403.2800*.

[Rabenseifner, 1999] Rabenseifner, R. (1999). Automatic mpi counter profiling of all users: First results on a cray t3e 900-512. In *Proceedings of the message passing interface developer's and user's conference*, volume 1999, pages 77–85.

[Ramos et al., 2013] Ramos, R. M., Martinello, M., and Rothenberg, C. E. (2013). Data center fault-tolerant routing and forwarding: An approach based on encoded paths. In *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*, pages 104–113. IEEE.

[Reich et al., 2013] Reich, J., Monsanto, C., Foster, N., Rexford, J., and Walker, D. (2013). Modular SDN programming with Pyretic. *Technical Reprot of USENIX*.

[Reitblatt et al., 2013] Reitblatt, M., Canini, M., Guha, A., and Foster, N. (2013). Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 109–114. ACM.

[Reitblatt et al., 2012] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. (2012). Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 323–334. ACM.

[Reitblatt et al., 2011] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. (2011). Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 7. ACM.

[Rekhter et al., 2006] Rekhter, Y., Li, T., and Hares, S. (2006). A Border Gateway Protocol 4 (BGP-4). RFC 4271, RFC Editor.

[Rémy, 2002] Rémy, D. (2002). Using, understanding, and unraveling the OCaml language from practice to theory and vice versa. In *Applied Semantics*, pages 413–536. Springer.

[Ren and Xu, 2014] Ren, T. and Xu, Y. (2014). Analysis of the new features of openflow 1.4. In *2nd International Conference on Information, Electronics and Computer*. Atlantis Press.

[Renner et al., 2015] Renner, T., Thamsen, L., and Kao, O. (2015). Network-aware resource management for scalable data analytics frameworks. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2793–2800. IEEE.

[Righi et al., 2016] Righi, R., Rodrigues, V., Costa, C., Galante, G., de Bona, L. C. E., and Ferreto, T. (2016). Autoelastic: Automatic resource elasticity for high performance applications in the cloud. *IEEE Transactions on Cloud Computing*, 4(1):6–19.

[Rubin et al., 2014] Rubin, E., Levy, E., Barak, A., and Ben-Nun, T. (2014). Maps: Optimizing massively parallel applications using device-level memory abstraction. *ACM Trans. Archit. Code Optim.*, 11(4):1–22.

[Ryu, 2015] Ryu (2015). Ryu sdn framework. Website. `http://osrg.github.io/ryu/`. Accessed 11 apr. 2015.

[Sakr et al., 2011] Sakr, S., Liu, A., Batista, D. M., and Alomari, M. (2011). A survey of large scale data management approaches in cloud environments. *Communications Surveys & Tutorials, IEEE*, 13(3):311–336.

[Salisbury, 2013] Salisbury, B. (2013). OpenFlow: Proactive vs Reactive Flows.

[Schäfer et al., 1996] Schäfer, M., Turek, S., Durst, F., Krause, E., and Rannacher, R. (1996). *Benchmark Computations of Laminar Flow Around a Cylinder*, pages 547–566. Vieweg+Teubner Verlag, Wiesbaden.

[Schlesinger et al., 2014] Schlesinger, C., Greenberg, M., and Walker, D. (2014). Concurrent NetCore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 11–24. ACM.

[Shan et al., 2000] Shan, H., Singh, J. P., Oliker, L., and Biswas, R. (2000). A comparison of three programming models for adaptive applications on the origin2000. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 11–11. IEEE.

[Sharan, 2012] Sharan, K. (2012). *Harnessing Java 7: A Comprehensive Approach to Learning Java*, volume 1. CreateSpace Independent Publishing Platform.

[Shaw, 1984] Shaw, M. (1984). Abstraction techniques in modern programming languages. *IEEE software*, (4):10–26.

[Sherwood et al., 2009] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. (2009). Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*.

[Shimonishi et al., 2012] Shimonishi, H., Takamiya, Y., Chiba, Y., Sugyo, K., Hatano, Y., Sonoda, K., Suzuki, K., Kotani, D., and Akiyoshi, I. (2012). Programmable network using OpenFlow for network researches and experiments. In *Proc. 6th International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2012)*, pages 164–171.

[Shin et al., 2013] Shin, S., Porras, P. A., Yegneswaran, V., Fong, M. W., Gu, G., and Tyson, M. (2013). FRESCO: Modular Composable Security Services for Software-Defined Networks. In *NDSS*.

[Simmons, 2014] Simmons, J. M. (2014). *Optical network design and planning*. Springer.

[Smith et al., 2014] Smith, M., Dvorkin, M., Laribi, Y., Pandey, V., Garg, P., and Weidenbacher, N. (2014). Opflex control protocol. *IETF, http://datatracker. ietf. org/doc/draft-smith-opflex*.

[Smolka et al., 2015] Smolka, S., Eliopoulos, S., Foster, N., and Guha, A. (2015). A Fast Compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM.

[Software Foundation, 2015] Software Foundation, P. (2015). Data Structures – Python 2.7.10 documentation. Website. https://docs.python.org/2/tutorial/datastructures.html. Accessed 10 aug 2015.

[Solano-Quinde et al., 2011] Solano-Quinde, L., Wang, Z. J., Bode, B., and Somani, A. K. (2011). Unstructured grid applications on gpu: performance analysis and improvement. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 13. ACM.

[Song, 2013] Song, H. (2013). Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA. ACM.

[Soulé et al., 2013a] Soulé, R., Basu, S., Kleinberg, R., Sirer, E. G., and Foster, N. (2013a). Managing the network with Merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 24. ACM.

[Soulé et al., 2014] Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. (2014). Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226. ACM.

[Soulé et al., 2013b] Soulé, R., Basu, S., Sirer, E. G., and Foster, N. (2013b). Scalable Network Management with Merlin.

[Soysal and Schmidt, 2010] Soysal, M. and Schmidt, E. G. (2010). Machine learning algorithms for accurate flow-based network traffic classification: Evaluation and comparison. *Performance Evaluation*, 67(6):451–467.

[Springer, 2011] Springer, P. (2011). Berkeley's dwarfs on cuda. *RWTH Aachen University, Tech. Rep*.

[Srivastava et al., 2016] Srivastava, G., Singh, M., Kumar, P., and Singh, J. (2016). Internet traffic classification: A survey. In *Recent Advances in Mathematics, Statistics and Computer Science*, pages 611–620. World Scientific.

[Stone et al., 2001] Stone, G. N., Lundy, B., and Xie, G. G. (2001). Network policy languages: a survey and a new approach. *Network, IEEE*, 15(1):10–21.

[Stuart et al., 2011] Stuart, M. B., Stensgaard, M. B., and Sparsø, J. (2011). The renoc reconfigurable network-on-chip: Architecture, configuration algorithms, and evaluation. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(4):45.

[Suzuki et al., 2014] Suzuki, K., Sonoda, K., Tomizawa, N., Yakuwa, Y., Uchida, T., Higuchi, Y., Tonouchi, T., and Shimonishi, H. (2014). A survey on OpenFlow technologies. *IEICE Transactions on Communications*, 97(2):375–386.

[Szyperski et al., 1999] Szyperski, C., Bosch, J., and Weck, W. (1999). Component-oriented programming. In *Object-oriented technology ecoop'99 workshop reader*, pages 184–192. Springer.

[Takahashi et al., 2015] Takahashi, K., Khureltulga, D., Munkhdorj, B., Kido, Y., Date, S., Yamanaka, H., Kawai, E., and Shimojo, S. (2015). Concept and design of sdn-enhanced mpi framework. In *Software Defined Networks (EWSDN), 2015 Fourth European Workshop on*, pages 109–110. IEEE.

[Takamiya, 2015] Takamiya, Y. (2015). trema. Website. `https://github.com/trema/trema`. Accessed 10 apr. 2015.

[Tang et al., 2017] Tang, F., Yang, L. T., Tang, C., Li, J., and Guo, M. (2017). A dynamical and load-balanced flow scheduling approach for big data centers in clouds. *IEEE Transactions on Cloud Computing*.

[Tangherloni et al., 2017] Tangherloni, A., Nobile, M. S., Besozzi, D., Mauri, G., and Cazzaniga, P. (2017). Lassie: simulating large-scale models of biochemical systems on gpus. *BMC bioinformatics*, 18(1):246.

[Tennenhouse et al., 1997] Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., and Minden, G. J. (1997). A survey of active network research. *Communications Magazine, IEEE*, 35(1):80–86.

[Thompson, 1999] Thompson, S. (1999). *Haskell: the craft of functional programming*, volume 2. Addison-Wesley.

[Tootoonchian et al., 2010] Tootoonchian, A., Ghobadi, M., and Ganjali, Y. (2010). OpenTM: traffic matrix estimator for OpenFlow networks. In *Passive and active measurement*, pages 201–210. Springer.

[Trestian et al., 2013] Trestian, R., Muntean, G. M., and Katrinis, K. (2013). Micetrap: Scalable traffic engineering of datacenter mice flows using openflow. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 904–907.

[Trois et al., 2017] Trois, C., de Bona, L. C. E., Fabro, M. D. D., Martinello, M., Bidkar, S., Nejabati, R., and Simeonidou, D. (2017). Softening up the network for scientific applications. In *Parallel, Distributed and Network-Based Processing (PDP), 25th Euromicro Inte. Conference on*, pages 410–418. IEEE.

[Trois et al., 2016a] Trois, C., de Bona, L. C. E., Fabro, M. D. D. D., and Martinello, M. (2016a). Carving Software-Defined Networks for Scientific Applications with SpateN. In *41st Conference on Local Computer Networks (LCN)*, pages 606–610. IEEE.

[Trois et al., 2016b] Trois, C., Fabro, M. D. D., de Bona, L. C. E., and Martinello, M. (2016b). A survey on sdn programming languages: Towards a taxonomy. *IEEE Communications Surveys Tutorials*, 18(4):2687–2712.

[Trois et al., 2015] Trois, C., Martinello, M., Bona, L., and Del Fabro, M. (2015). From Software Defined Network To Network Defined for Software. In *Proceedings of the 2015 ACM Symposium on Applied Computing*. ACM.

[U-Chupala et al., 2014] U-Chupala, P., Ichikawa, K., Iida, H., Kessaraphong, N., Uthayopas, P., Date, S., Abe, H., Yamanaka, H., and Kawai, E. (2014). Application-oriented bandwidth and latency aware routing with open flow network. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 775–780.

[Ucoluk and Kalkan, 2012] Ucoluk, G. and Kalkan, S. (2012). *Introduction to programming concepts with case studies in Python*. Springer Science & Business Media.

[Van Adrichem et al., 2014] Van Adrichem, N. L., Doerr, C., Kuipers, F., et al. (2014). Opennetmon: Network monitoring in openflow software-defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–8. IEEE.

[Van Deursen et al., 2000] Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36.

[Van Roy et al., 2009] Van Roy, P. et al. (2009). Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104.

[Vapnik, 1999] Vapnik, V. (1999). *The Nature of Statistical Learning Theory*. Springer Science & Business Media.

[Vassiliev, 2017] Vassiliev, A. V. (2017). Scalable opencl fpga computing evolution. In *Proceedings of the 5th International Workshop on OpenCL*, page 22. ACM.

[Vassoler et al., 2012] Vassoler, G. L., De Souza, F. R., Pedro Filho, P., and Ribeiro, M. R. (2012). Hybrid reconfiguration for upgrading datacenter interconnection topology. In *IEEE Photonics Conference 2012*.

[Veiga Neves et al., 2014] Veiga Neves, M., De Rose, C. A., Katrinis, K., and Franke, H. (2014). Pythia: Faster big data in motion through predictive software-defined network optimization at runtime. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 82–90. IEEE.

[Vencioneck et al., 2014] Vencioneck, R., Vassoler, G., Martinello, M., Ribeiro, M., and Marcondes, C. (2014). FlexForward: Enabling an SDN manageable forwarding engine in Open vSwitch. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 296–299.

[Vetter and Mueller, 2003] Vetter, J. S. and Mueller, F. (2003). Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865.

[Voellmy et al., 2010] Voellmy, A., Agarwal, A., and Hudak, P. (2010). Nettle: Functional reactive programming for openflow networks. Technical report, DTIC Document.

[Voellmy et al., 2012] Voellmy, A., Kim, H., and Feamster, N. (2012). Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 43–48. ACM.

[Voellmy and Wang, 2012] Voellmy, A. and Wang, J. (2012). Scalable software defined network controllers. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 289–290. ACM.

[Voellmy et al., 2013] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. (2013). Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 87–98. ACM.

[Wan and Hudak, 2000] Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. In *ACM SIGPLAN Notices*, volume 35, pages 242–252. ACM.

[Wang et al., 2012a] Wang, G., Ng, T., and Shaikh, A. (2012a). Programming your network at run-time for big data applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 103–108. ACM.

[Wang et al., 2016] Wang, J., Rubin, N., Sidelnik, A., and Yalamanchili, S. (2016). Laperm: Locality aware scheduler for dynamic parallelism on gpus. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 583–595. IEEE Press.

[Wang et al., 2012b] Wang, K., Qi, Y., Yang, B., Xue, Y., and Li, J. (2012b). LiveSec: Towards effective security management in large-scale production networks. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 451–460. IEEE.

[Wang et al., 2008] Wang, N., Ho, K., Pavlou, G., and Howarth, M. (2008). An overview of routing optimization for internet traffic engineering. *Communications Surveys & Tutorials, IEEE*, 10(1):36–56.

[Wang et al., 2011] Wang, R., Butnariu, D., Rexford, J., et al. (2011). Openflow-based server load balancing gone wild. *Hot-ICE*, 11:12–12.

[Werner et al., 2017] Werner, S., Navaridas, J., and Luján, M. (2017). Designing low-power, low-latency networks-on-chip by optimally combining electrical and optical links. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 265–276. IEEE.

[Whalen et al., 2012] Whalen, S., Engle, S., Peisert, S., and Bishop, M. (2012). Network-theoretic classification of parallel computation patterns. *International Journal of High Performance Computing Applications*, page 1094342012436618.

[Wu et al., 2016] Wu, Z., Lu, K., Wang, X., and Chi, W. (2016). Alleviating network congestion for hpc clusters with fat-tree interconnection leveraging software-defined networking. In *Systems and Informatics (ICSAI), 2016 3rd International Conference on*, pages 808–813. IEEE.

[Xorplus, 2017] Xorplus, P. (2017). Pica8 Xorplus. Website. `http://sourceforge.net/projects/xorplus/`. Accessed 11 May 2017.

[Yamanaka et al., 2014] Yamanaka, H., Kawai, E., Ishii, S., and Shimojo, S. (2014). Autovflow: Autonomous virtualization for wide-area openflow networks. In *Third European Workshop on Software Defined Networks*.

[Yan et al., 2017] Yan, J., Liu, X., and Jin, D. (2017). Simulation of a software-defined network as one big switch. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 149–159. ACM.

[Yao et al., 2014] Yao, F., Wu, J., Venkataramani, G., and Subramaniam, S. (2014). A comparative analysis of data center network architectures. In *International Conference on Communications (ICC)*, pages 3106–3111. IEEE.

[Yao et al., 2016] Yao, H., Li, H., Liu, C., Xiong, M., Zeng, D., and Li, G. (2016). Joint optimization of vm placement and rule placement towards energy efficient software-defined data centers. In *Computer and Information Technology (CIT), 2016 IEEE International Conference on*, pages 204–209. IEEE.

[Yin et al., 2014] Yin, C., Kuo, T.-C., Li, T.-Y., Chang, M.-C., and Liao, B.-H. (2014). Mediating between openflow and legacy transport networks for bandwidth on-demand services. In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pages 1–4. IEEE.

[Yu et al., 2004] Yu, F., Katz, R. H., and Lakshman, T. V. (2004). Gigabit rate packet pattern-matching using tcam. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 174–183. IEEE.

[Yu et al., 2017] Yu, J., Liu, G., Dong, W., and Li, X. (2017). Using locality-enhanced distributed memory cache to accelerate applications on high performance computers. In *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing,(HPSC) and IEEE International Conference on Intelligent Data and Security (IDS), 2017 IEEE 3rd International Conference on*, pages 160–166. IEEE.

[Yu et al., 2014] Yu, Y., Qian, C., and Li, X. (2014). Distributed and collaborative traffic monitoring in software defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 85–90. ACM.

[Zaalouk et al., 2014] Zaalouk, A., Khondoker, R., Marx, R., and Bayarou, K. (2014). OrchSec: An orchestrator-based architecture for enhancing network-security using Network Monitoring and SDN Control functions. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9. IEEE.

[Zhang et al., 2012] Zhang, H., Lu, G., Qassrawi, M. T., Zhang, Y., and Yu, X. (2012). Feature selection for optimizing traffic classification. *Computer Communications*, 35(12):1457–1471.

[Zhang and Ge, 2005] Zhang, Y. and Ge, Z. (2005). Finding critical traffic matrices. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 188–197. IEEE.

[Zhao et al., 2013] Zhao, Y., Jia, W., Hu, R.-X., and Min, H. (2013). Completed robust local binary pattern for texture classification. *Neurocomputing*, 106:68 – 76.