FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

DESIGN AND EVALUATION OF TECHNIQUES FOR HPC PLATFORMS WITH SDN-CAPABLE INTERCONNECTS

By

SAPTARSHI BHOWMIK

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2025

Saptarshi Bhowmik defended this thesis on July, 2025.
The members of the supervisory committee were:

Xin Yuan
Professor Directing Thesis

Vanessa Dennen
University Representative

Weikuan Yu
Committee Member

Gary Tyson
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

I gratefully acknowledge the guidance of my advisor, Prof. Xin Yuan, and the steadfast support of my committee members—Prof. Weikuan Yu, Prof. Gary Tyson, and Prof. Vanessa Dennen. I am indebted to Prof. Fengfeng Ke, whose continual mentorship has been invaluable throughout my graduate studies. I also thank my colleagues in the HPC research group for our many fruitful discussions. Most of all, I am thankful to my mother and father for their unwavering love, constant encouragement, and enduring faith in my abilities.

This material is based upon work supported by the National Science Foundation (NSF) under Grants CRI1822737 and SHF2007827. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), supported by NSF grant ACI1548562; the XSEDE "Bridges" resource at the Pittsburgh Supercomputing Center (PSC) through allocations ECS190004 and CCR200042; and the "Bridges2" resource at PSC through allocation CIS230062 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The introduction of GPU-based compute nodes alters the balance between computation and communication aspects within the system, shifting the communication-to-computation ratio. As computation speeds up with the addition of these powerful nodes, communication fails to scale proportionally. New networking technologies like Software-Defined Networking (SDN) address this problem by providing improved resource management during communication. This dissertation proposes to leverage SDN to enhance communication efficiency in HPC environments with GPU-based compute nodes and to find efficient system configurations for running HPC applications in these environments. By alleviating communication bottlenecks, this research aims to facilitate seamless execution of HPC workloads on SDN-based HPC systems and thereby enable groundbreaking discoveries in scientific computing.

CHAPTER 1

INTRODUCTION

Software Defined Networking (SDN) [38] has emerged as a promising technology and has been widely implemented across various network environments, including data centers, campus networks, and wide-area networks. SDN offers several notable features: (1) a centralized global network view for intelligent traffic and resource management, (2) flexible per-flow management to accommodate varying network traffic patterns, (3) network monitoring capabilities providing valuable traffic statistics, and (4) ease of integrating new network functionalities and services. These features empower SDN to effectively manage traffic at the flow level using a centralized view and optimize network resource utilization for improved performance compared to traditional networking infrastructures [8]. While these SDN capabilities hold appeal for High-Performance Computing (HPC) systems and applications, SDN adoption within the HPC domain remains limited. One reason for this is the absence of clear evidence demonstrating SDN's superiority over existing networking technologies in high-end HPC systems that employ sophisticated routing schemes. Existing SDN methodologies are primarily tailored for internet and data-parallel applications like Hadoop and MapReduce applications [28], which have communication characteristics different from those of HPC applications. Consequently, to achieve optimal performance on SDN-based HPC systems, novel techniques that consider the unique communication patterns of HPC applications are of utmost importance.

Large-scale systems aiming to achieve over 1 Exaflop/s of sustained performance have been built. Unlike the systems dominating the HPC industry a decade ago, many of today's and future systems consist of a relatively modest number of nodes. For instance, Sequoia at LLNL [43], the fastest supercomputer in the Top500 list in 2012, utilized 96K nodes to achieve 20 Petaflop/s of peak performance. In comparison, Summit at ORNL [51], one of the fastest supercomputers as of June 2020, employs approximately 4600 nodes but achieves a peak performance of 200 Petaflop/s. The driving force behind the reduction in the number of nodes is compute acceleration devices such as GPUs [52]. For example, an NVIDIA Volta V100 can perform 7 Teraflop/s worth of double-precision computation compared to 200 Gigaflop/s for a Blue Gene/Q node. However, such a significant increase in computing capability has not been matched by a similar increase in network capability. Additionally, the communication performance achievable by an application

on a system remains the major bottleneck for the overall application performance. Therefore, while communication needs tend to expand at a slower rate compared to computational demands (e.g., analogous to the growth of surface area versus volume), it remains vital to determine the optimal utilization of existing communication capabilities and achieve an ideal balance between computation and communication capabilities. The central inquiry we aim to address is whether a system featuring fewer nodes, each possessing greater computing capability, outperforms a system comprising more nodes, each with lesser computing capability.

My dissertation research primarily focuses on the following two areas:

1. Developing and evaluating SDN techniques for HPC systems. I am investigating the influence of SDN on HPC environments, whether SDN can improve the communication performance for HPC systems, and whether our proposed techniques result in higher communication performance in SDN-capable interconnect topologies than existing schemes.

2. Exploring the performance of modern HPC systems across diverse network onfigurations. I am using end-to-end system simulations to explore the performance impact of various network design and parameter choices for GPU-based systems, conducting a sensitivity study of the overall performance with respect to changes in these parameters.

The structure of this dissertation aligns with the outlined research objectives. Chapter 2 will provide a background on interconnection technologies, SDN fundamentals and related works. In Chapter 3, I describe the implementation of various SDN-based enhancements in HPC environments and evaluate their impact on application performance. Chapter 4 examines how key network parameters influence modern HPC systems. Finally, chapter 5 will offer concluding remarks.

CHAPTER 2

BACKGROUND AND RELATED WORKS

In the domain of High-Performance Computing (HPC) and data center networks, the coordination of numerous hardware components is crucial for them to function as a unified system. This coordination happens through an interconnection network, which serves as the backbone for communication among these components. Thousands of hardware pieces collaborate over this interconnection network to ensure smooth operation. The effectiveness of this interconnect relies on various design choices, such as the topology used to connect physical components, the routing scheme to select communication paths, and managing network traffic loads along these paths and links. In this chapter, I provide essential background information on topology, routing and Software Defined Networks (SDN). These fundamental concepts, will give readers the insight into how hardware components interact and how interconnect designs can be optimized for better performance within HPC and data center environments.

2.1 Topology

The interconnect network is often represented as a graph, where each node represents a piece of hardware like a server or a switch, and each line (edge) represents a link between them. Servers are referred to as processing elements, while switches and routers are referred to as forwarding elements. Two nodes directly connected by a link are neighbors. The number of links a node has is called its nodal degree [14]. The distance between two nodes is how many links (or hops) it takes to get from one to the other [14]. The diameter of a network is the longest distance between any two nodes. Splitting the network into two equal halves is called a bisection, and the bandwidth of this split is how much data can flow between the two halves without slowing down [14]. The bisection bandwidth is the lowest possible bandwidth among all possible splits. In HPC and data-center networks, low diameter and high bisection bandwidth are essential for performance, whereas a low nodal degree reduces costs and simplifies the design [40]. To meet these challenges, different types of topologies are used. In data centers, one of the most commonly utilized interconnection topologies is the fat-tree. This design has gained popularity due to its ability to efficiently provide a high

throughput and low latency for communication. For bigger systems, like exascale supercomputers, the dragonfly topology has been gaining popularity recently. It's designed to be scalable and cost-effective on a large scale. As technology evolves, new topologies will likely be developed to meet the demands of future interconnects.

2.1.1 Fat-tree

Fat-tree topology represents a robust architecture for high-performance computing environments, characterized by its hierarchical structure and abundant bandwidth allocation [39]. In this topology, switches and compute nodes are organized into a tree like structure, with bandwidth increasing as one ascends toward the root of the tree. In a typical fat-tree setup, such as the 3-level full bisection bandwidth fat-tree, switches are classified into three categories:

- **Core Switches :** These switches reside at the highest layer and serve to interconnect different pods [19].
- **Aggregate Switches :** Positioned between the core and leaf switches, aggregate switches link to the leaf switches within a pod, forming a cohesive unit.
- **Leaf Switches :** Located at the bottom layer, leaf switches interface directly with the compute nodes, facilitating communication within the pod.

Figure 2.1 shows a example of a three level fat-tree.

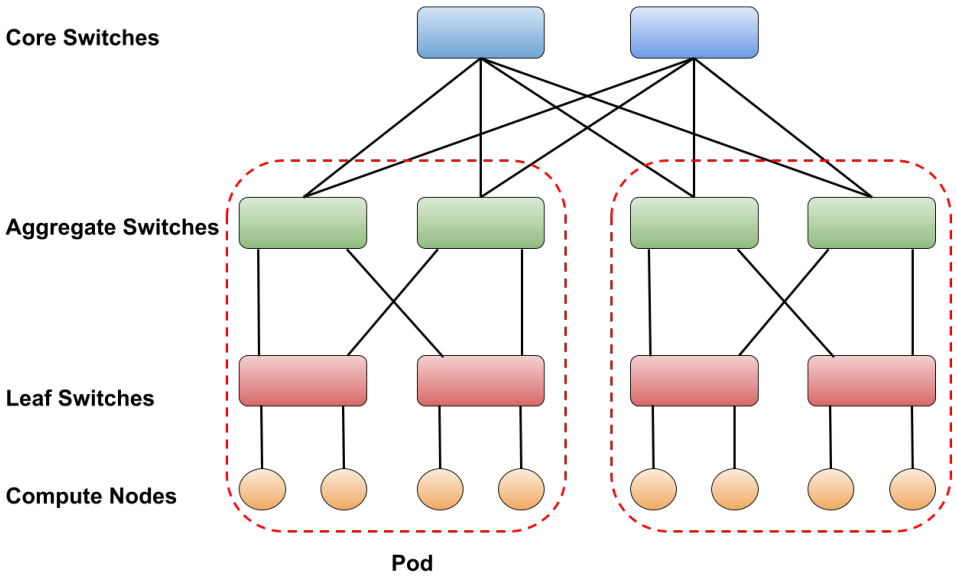

Figure 2.1: A three level fat-tree with core, aggregate, leaf switch levels and pods

Nearly every practical fat-tree topology can be expressed through the extended generalised fat-tree (XGFT) notation [49]. An XGFT is a parameterised model for fat-tree interconnects defined as

$$\mathrm{XGFT}\big(h; m_1, \ldots, m_h; w_1, \ldots, w_h\big),$$

where h is the height of the tree, a switch at level i has m_i downward links to children and w_i upward links to parents. Compute nodes attach at level 0, and the total number of nodes is

$$N \;=\; \prod_{i=1}^{h} m_i.$$

Figure 2.2 shows, in four steps, how to build a three-level fat-tree with XGFT notation. First, start with a single node, since there no links and the height of the tree is 0, the XGFT notation is XGFT(0; ;). Second, connect four of these nodes with one switch, which is also called leaf switch; this makes XGFT(1; 4; 1), where every node has one link going up and the leaf switch has four links going down. Third, create four copies of that one-level tree and join the leaf switches to two aggregate switches, giving XGFT(2; 4, 4; 1, 2), here 4 links go down from each aggregate switch to leaf switches, and two links go up from each leaf switch to connect to the aggregate switches. Last, connect three of these two-level trees to four core switches to form the three level fat-tree. Since each core switch has 3 downlinks and each aggregate switch has 2 uplinks, the XGFT notation is XGFT(3; 4, 4, 3; 1, 2, 2).

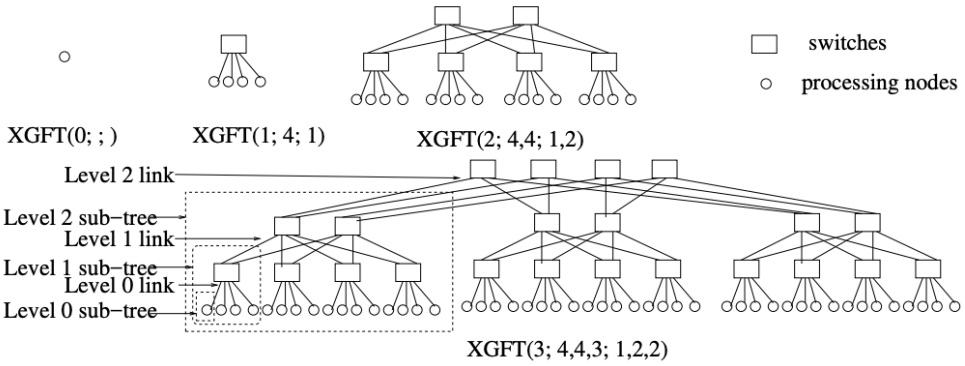

Figure 2.2: A fat-tree represented in XGFT.

A fat-tree is said to provide full bisection bandwidth when every internal switch offers as many upward links as there are downward links feeding into it [54], i.e.,

$$w_i = m_{\,i-1} \qquad \text{for all } i = 1, \ldots, h.$$

This 1:1 balance of ingress and egress capacity at each switch means packets that arrive from the layer below can be forwarded upward without blocking. Because each level preserves the available bandwidth, the minimal cut that divides the end hosts into two equal halves of nodes has a combined capacity matches the total injection bandwidth of either half, thereby guaranteeing full bisection throughput. In a three-level fat-tree, for instance, $m_1 = w_2$ and $m_2 = w_3$ uphold this property.

One method used to reduce the cost of building a fat-tree network is tapering [31]. Tapering involves connecting more devices to each switch at the lower levels of the network. While this may decrease the total bandwidth available at higher levels, it also reduces the number of switches and cables needed to connect the same number of devices compared to a full bisection fat-tree. A tapered fat-tree is obtained by provisioning fewer uplinks than downlinks at one or more switch levels. In XGFT notation a tree is tapered if $w_i < m_{i-1}$ for at least one $i \in \{1, \ldots, h\}$. The tapering ratio is the ratio of number of downlinks to the number of uplinks in a switch at a given level, it determines the extent of uplink reduction [31, 63]. In a 3-to-1 tapering configuration, for every one uplink from a leaf switch, three downlinks connect to three compute nodes each. The figure below (Figure 2.3) illustrates a tapered fat-tree pod for a 3-level fat-tree with 32 port switches.

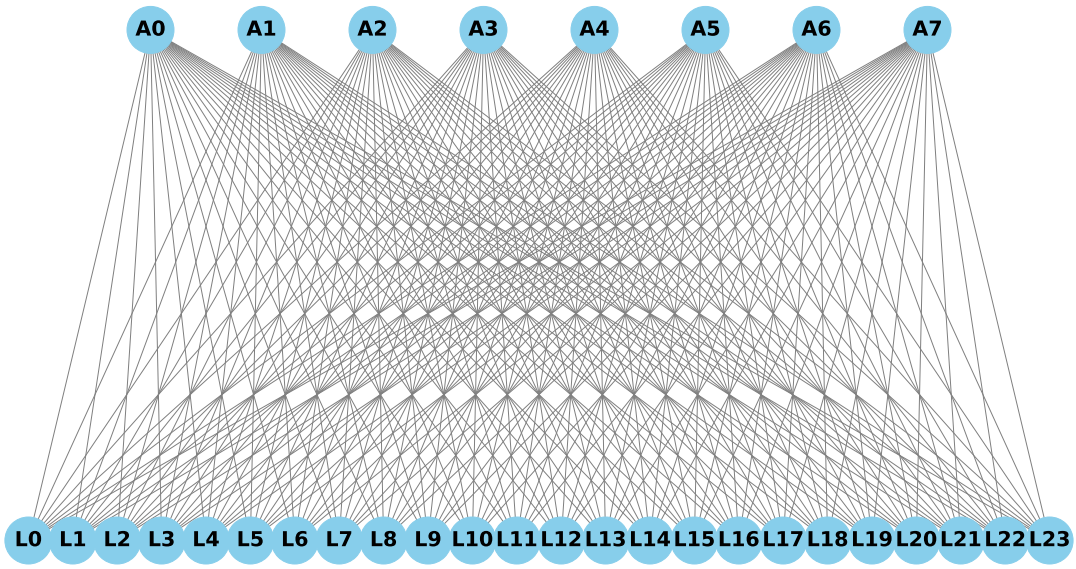

Figure 2.3: 3-to-1 Tapered Fat-tree pod

2.1.2 Dragonfly

The dragonfly topology stands out as a cost-effective solution for building expansive interconnection networks [35]. This design is characterized by its two-layer structure, exemplified in Figure 2.4. Initially proposed by Kim et al. [35], the dragonfly topology employs a multi-level dense configuration, primarily leveraging high-radix routers. In its basic form, a dragonfly network comprises interconnected routers forming groups, each resembling a virtual router with a notably high radix [8]. These groups are then interconnected through an inter-group topology. Figure 2.4 illustrates a sample dragonfly network with nine groups, each containing four switches. There are variations of dragonfly topology, including canonical dragonfly, hamming dragonfly, and dragonfly plus, which utilize various intra-group connectivity patterns [27]. However, all implementations of dragonfly topology feature all-to-all connectivity between groups.

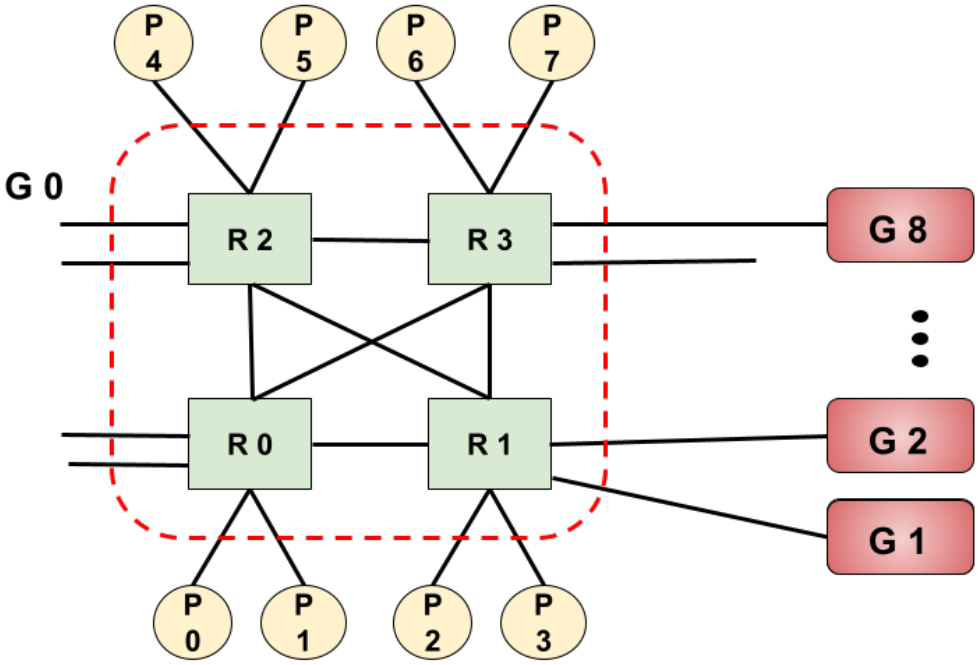

Figure 2.4: Dragonfly architecture with 9 groups and 4 routers per group

Four fundamental parameters characterize the dragonfly topology: the number of compute nodes in each switch (p), the intra-group links per switch (a), the inter-group links per switch (h), and the total number of groups (g) [35]. The dragonfly network comprises of $g \times a$ routers, and $g \times a \times p$ compute nodes, where g represents the number of groups, a signifies the intra-group links per switch, and p denotes the compute nodes per switch. Each group has $a \times h$

global links, with h representing the inter-group links per switch. Notably, the maximum size dragonfly configuration is characterized by $g = a \times h + 1$ groups. Several prominent supercomputer architectures, such as the Cray Cascade architecture and the Cray Slingshot network, have embraced variations of the dragonfly topology [16, 15]. These implementations have found their way into current supercomputers like Titan [50] and Trinity [7], as well as future exascale computing designs. In summary, the dragonfly topology, with its versatile structure and efficient utilization of high-radix routers, presents a compelling solution for constructing large-scale interconnection networks, offering both cost-effectiveness and scalability.

2.2 Routing

Routing in an interconnect topology determines how data packets, which are units of data transmitted over a network, travel from a source to an destination within a network. Each data packet contains information such as the source address, destination address, and the actual data being transmitted. These packets are sent from a source node to a destination node, with each node being connected to a switch. The source switch connects to the source node, while the destination switch connects to the destination node. It involves mapping network flows to specific paths for data transmission; a process influenced by the network's underlying topology. The efficiency of routing directly impacts the performance of applications in HPC and data center systems, making it a crucial aspect of interconnect modeling. Routing in interconnect networks can be classified into three broad categories: when the routing decision is taken, how the path is selected, and where the decision is taken [60, 14]. In terms of the timing at which the routing decision is made, there are offline schemes and online schemes; offline schemes establish routes statically during system configuration, whereas online schemes compute routes at runtime to mitigate congestion. In terms of path selection, there are deterministic routing, oblivious routing, and adaptive routing. Deterministic routing assigns a single, immutable path to each source–destination node pair; oblivious routing chooses among multiple paths without considering current network conditions; and adaptive routing uses real-time network indicators such as queue occupancy or link utilisation to select a path from a set of paths. Finally, in terms of the location at which the decision is made, there are source routing and distributed routing. In source routing, the entire path is encoded in the packet header at the source node, whereas in distributed routing each intermediate node along the path makes routing decisions based on local routing tables or forwarding rules. These routing strategies play a vital role in ensuring efficient data transmission in HPC and data center environments.

2.2.1 Routing in Fat-tree

Routing strategies within fat-tree networks can be categorized as either oblivious or adaptive to network communication traffic. In fat-tree networks, oblivious routing algorithms consistently select the same nearest common ancestor (NCA) for all communication between a given source-destination pair. These routing paths can be pre-computed and stored in forwarding tables or calculated dynamically based on simple formulas using source and destination labels. Two common static oblivious routing schemes in fat-tree networks are source-mod-k (S-mod-k) and destination-mod-k (D-mod-k). In S-mod-k at each upward hop, the switch selects port ($s \bmod k$), where s is the numeric identifier of the source node and k is the number of uplinks from the current switch. Similarly, in D-mod-k, the switch selects port ($d \bmod k$), where d is the numeric identifier of the destination node [59, 46]. In both these routing, after the packet reaches the NCA, it travels downward along the unique path to the destination node. Although, these routing algorithms are widely used in many HPC systems as the default static routing algorithm, they may exhibit poor performance for both average and worst-case permutation traffic patterns.

Adaptive routing dynamically select paths based on the current network state, such as link congestion or available bandwidth, to optimize performance in real-time. During the upward direction toward the nearest common ancestor (NCA) for both the source and destination routers, adaptive routing algorithms prioritize forwarding packets to the least congested port available [23]. This approach helps to alleviate congestion and optimize the utilization of network resources by steering traffic along paths with ample capacity. Once the packet reaches the NCA, which acts as the central router for the source-destination pair, it is then directed along a unique downward path toward the destination router. By dynamically adapting routing paths based on real-time network conditions, adaptive routing enhances the efficiency and performance of fat-tree networks. Routing decisions in fat-tree networks typically focus on determining the upward paths to carry traffic for each source-destination pair.

2.2.2 Routing in Dragonfly

In a dragonfly topology, the source node belongs to the source group, and the destination node belongs to the destination group. Traffic packets between these nodes can travel along either a minimal or a non-minimal path. Broadly speaking, the dragonfly network has three popular routing schemes.

The minimal routing scheme (MIN) proposed by Kim *et al.* [35] always forwards a packet along the shortest path between a source router R_s and a destination router R_d in a dragonfly topology. If R_s and R_d lie in the same group or both of them are connected by a global link, the path is a single hop, if one of the router is connected to the global link which goes to the other's group then it is a two-hop path, the packet never travels more than one global link. A minimal path comprises no more than three hops:

1. a local hop inside the source group to a gateway router R_a that owns a global link to the destination group;
2. a single global hop from R_a to a gateway router R_b in the destination group;
3. a local hop from R_b to the destination router R_d.

Its deterministic nature, however, makes it vulnerable to adversarial permutations such as the shift pattern, in which every router of a source group simultaneously send to the same destination group as such the global link between the two groups is shared by all routers to send its data, and it becomes a bottleneck.

Valiant Load-Balanced Routing (VLB) [35, 33], sends each packet through a randomly chosen intermediate router in an intermediate group before it reaches its destination. This detour spreads traffic evenly across the fabric and removes congestion that can happen when many packets tries to use the same global link to reach their destination. A packet routed in VLB path can traverse up to two global links and no more than five hops:

1. a local hop inside the source group from the source router R_s to a gateway router R_a that has a global link to the intermediate group;
2. a global hop from R_a to a gateway router R_b in the intermediate group;
3. a local hop within the intermediate group from R_b to another gateway router R_c that owns a global link to the destination group;
4. a second global hop from R_c to a gateway router R_t in the destination group; and
5. a final local hop from R_t to the destination router R_d.

By randomising the choice of the intermediate group, VLB balance the network load in adversarial traffic patterns, at the cost of the extra distance introduced by the detour.

Universal Globally-Adaptive Load-balancing (UGAL) is an adaptive routing algorithm that dynamically chooses, for every packet, between a MIN path and a VLB path [35]. It monitors the

amount of data packets queued up or waiting to be transmitted at the source router. By assessing the level of queue occupancy, it can infer the current state of congestion within the network. If the queue occupancy is high, there can be congestion, and it may opt for routing strategies that help alleviate congestion, such as selecting VLB paths to distribute traffic more evenly across available links.

Two practical variants UGAL are: *UGAL-G* which assumes perfect, system-wide knowledge of queue occupancies and is used mainly as an upper-bound benchmark. *UGAL-L*, which relies only on local queue occupancy information at the source router [35]. Let q_m and q_{nm} denote the output-queue occupancies of the first link on the minimal and non-minimal paths, and let H_m and H_{nm} be the respective hop counts. UGAL-L approximates the latency of each path as the product of these two quantities and introduces a bias b that tilts the decision toward MIN path or the VLB path [35]. A packet is routed minimally only if

$$q_m \times H_m \ \leq \ q_{nm} \times H_{nm} + b,$$

otherwise it is routed non-minimally.

2.3 HPC Applications

In high-performance computing, workloads comprise real applications, which address substantive scientific, engineering, finance and other real-world problems, and proxy applications, which imitate the computational and communication characteristics of real applications in a reduced form. Moreover, system architects employ synthetic traffic patterns, which are artificially constructed sequence of message transfers, to evaluate performance under carefully controlled conditions. Many HPC applications share a common iterative structure, each iteration entails a sequence of computational tasks executed in parallel, followed by communication and synchronization steps. These iterations form the backbone of many HPC workflows, where complex computations are distributed across vast computational resources. As data flows through the system, results are exchanged, aggregated, and synchronized to drive iterative refinement and convergence.

In my work I have chosen a set of benchmarks which is represents the diverse HPC workloads along with some synthetic traffic, to ensure that the performance evaluation of my algorithms is comprehensive and applicable across a wide array of HPC scenarios. The following is a brief description of the applications I used in this work:

- **Random permutation**: A synthetic traffic where each node sends a message to another randomly chosen node. The source destination pair is unique across the whole permutation.
- **Stencil3d**: MPI benchmark with 6-point near-neighbor communication in a 3D virtual process grid.
- **Stencil4d**: MPI benchmark with 8-point near-neighbor communication in a 4D virtual process grid.
- **Random mixed**: A synthetic traffic which is composed of two different random permutation.
- **Stencil mixed**: MPI benchmark which is composed of stencil3d kernel followed by stencil4 kernel together.
- **Random permutation third**: A synthetic traffic where only a third of the flows in random permutation is used.
- **Subcomm3d**: MPI benchmark with all-to-all communication within subsets of processes in a 3D virtual process grid.
- **Kripke**: 3D S^n deterministic particle transport code, which runs an MPI-based parallel sweep algorithm [41].
- **Laghos**: Proxy application that solves time-dependent Euler equations with MPI-based domain decomposition [44].
- **AMG**: Parallel algebraic multigrid solver [56].
- **SW4lite**: Proxy application for SW4 [61], a 3D seismic modeling code.
- **Lammps**: LAMMPS is an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator, it solves equations of motion for a collection of interacting particles. It partitions the simulation domain into small sub-domains to solve a problem [66].
- **Nekbone**: Solves 3D Poisson problem in rectangular geometry. The key MPI operations are matrix-matrix multiplication, inner products, nearest neighbor communication, MPI_Allreduce [24].
- **MILC**: Performs four dimensional SU(3) lattice gauge theory, mainly through near-neighbor communication and MPI_Allreduce [25].

While evaluating these diverse HPC benchmarks, it is essential to understand the communication characteristics that fundamentally influence their performance and scalability. HPC applications typically exhibit two primary communication characteristics: regular (static) and irregular (dynamic and dynamically analyzable) communication patterns. Recognizing these patterns allows for more precise optimization of network resources, thereby enhancing overall efficiency and performance.

- **Regular communication:** Regular communication in HPC applications involves predictable and repetitive data exchanges that are determined before the execution of the application. This type of communication includes static communication patterns where the source and destination nodes, as well as the message sizes, are all known during compile time. Figure 2.5 displays an MPI code segment of Stencil4d, a representative HPC kernel. This kernel performs nearest neighbor communication: each process communicates with its 8 neighbors (front/back, top/bottom, left/right, and north/- south) in the 4-dimension domain with 8 MPI Isends and 8 MPI Irecvs. The figure only shows one MPI Isend and one MPI Irecv since the others are similar. The communicator MPI COMM WORLD remains fixed during the execution, and the source and destination, in this case process north of the present node, as well as the message size are known before the application executes. Hence, these communications are static. Most communications in HPC applications are static, allowing for more straightforward optimization and improved overall performance by leveraging the predictability of these communication patterns.

- **Irregular communication:** In HPC applications, communication is irregular when the data-transfer pattern cannot be determined prior to program execution. Most irregular communications are dynamic or dynamically analyzable. In dynamically analyzable communication, the source and destination nodes , as well as message sizes, can be determined at runtime but not during compile time. For dynamic communication, these parameters cannot be determined until after the communication has occurred. Figure 2.6 displays an MPI code segment of the primary solver function for Laghos. In Laghos, a new communicator is established each time the function is called, and communications are performed in that communicator as can be seen in Line 3 of the figure. The communication information for this application is unknown until the communication is performed and the communications are thus dynamic.

```
for (int i = 0; i < MAX_ITER; i++) {
        MPI_Isend(sendn, 100000000, MPI_CHAR, north, 9,
           MPI_COMM_WORLD, &reqs[0]
);
        ...
        MPI_Irecv(recvn, 100000000, MPI_CHAR, north, 9,
           MPI_COMM_WORLD, &reqs[8]
);
        ...
        MPI_Waitall(16, req status);
}
```

Figure 2.5: Stencil4d code snippet

```
LagrangianHydroOperator (...){
        ...
        ParMesh *pm = H1FESpace.GetParMesh();
        MPI_Allreduce(&loc_area, &glob_area, 1, MPI_DOUBLE, MPI_SUM,
            pm->GetComm
());
        ...
}
```

Figure 2.6: Laghos code snippet

2.4 Software Defined Network

Software Defined Network(SDN) is a modern networking scheme where, the organization of network functionality is often conceptualized into three distinct layers: the data plane, the control plane, and the management plane [38]. Each layer serves a critical role in facilitating the efficient operation and management of the network infrastructure. Figure 2.7 shows the three planes in the realm of SDN abstraction.

- **Data Plane:** The data plane, also known as the forwarding plane, is responsible for the actual transmission of data packets within the network [8]. It consists of networking devices such as routers, switches, and other forwarding elements. These devices receive incoming packets and make forwarding decisions based on predetermined rules or protocols. The primary function of the data plane is to ensure that data packets are correctly routed to their intended destinations across the network.

- **Control Plane:** The control plane is tasked with managing the forwarding and routing mechanisms within the network [8]. It determines how data packets should be forwarded based on factors such as network topology, traffic conditions, and routing policies. Traditionally, the control plane functions are embedded within the networking devices themselves, leading to a tightly coupled architecture where control and data planes operate in conjunction with each other. This tightly integrated approach has been essential for ensuring network resilience and stability, particularly in large-scale distributed networks such as the Internet.

- **Management Plane:** The management plane oversees the overall management and configuration of the network infrastructure [8]. It is responsible for tasks such as network monitoring, configuration management, performance optimization, and security policy enforcement. The management plane provides administrators with the tools and interfaces necessary to manage and control various aspects of the network, ensuring its reliability, security, and efficiency.

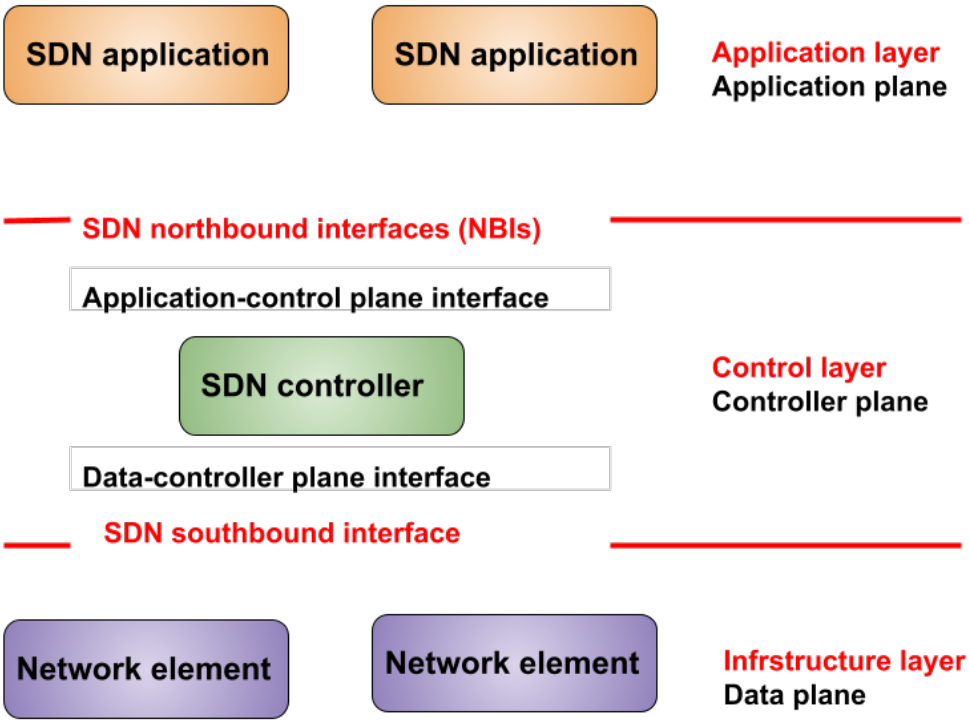

Figure 2.7: SDN abstraction

While the traditional networking architecture with tightly coupled control and data planes has been successful in ensuring network resilience, it also poses several limitations. One of the primary challenges is the complexity and rigidity of the architecture, which makes it difficult to introduce innovations and adapt to changing network requirements. Additionally, the decentralized nature of the control plane makes it challenging to achieve a holistic view of the network, hindering effective management and optimization. To address these limitations and enable greater flexibility and agility in network management, the concept of Software Defined Networking (SDN) has emerged as a promising approach. SDN decouples the control plane from the data plane, allowing for centralized management and programmability of the network. In an SDN architecture, the control logic is moved to a centralized entity known as the controller or Network Operating System (NOS), which maintains a global view of the network and is responsible for configuring forwarding policies.

The key components of an SDN architecture include the following:

- **Decoupled Data and Control Planes:** By separating the control logic from the underlying networking devices, SDN enables greater flexibility, scalability, and agility in network

management. It allows administrators to dynamically adjust network behavior in response to changing traffic patterns and application requirements, leading to improved performance and resource utilization [8, 38].

- **Centralized Controller:** The centralized controller serves as the brain of the SDN architecture, maintaining a global view of the network and orchestrating the forwarding policies for all connected devices. The controller communicates with the networking devices via standardized protocols such as OpenFlow, providing a centralized point of control for the entire network [8, 38].
- **Programmable Network Behavior:** One of the key advantages of SDN is its programmability, which enables administrators to implement innovative networking services and applications through software applications running on top of the SDN controller. This programmability allows for the dynamic creation and deployment of network policies, enabling administrators to tailor the network behavior to specific application requirements and business needs [8, 38].

In recent years, SDN has gained widespread acceptance and adoption in both industry and research communities. Its flexibility and programmability have led to a wide range of applications across various domains, including data centers, telecommunications, and cloud computing [3, 17]. One area of particular interest is the application of SDN in high-performance computing (HPC) environments. HPC systems often require fast and efficient communication between compute nodes to handle large-scale scientific computations and data-intensive workloads. By leveraging SDN, researchers aim to optimize routing and topologies in HPC environments, improving communication efficiency and resource utilization.

The structure of an SDN based HPC System (SHS) is depicted in Figure 2.8. The SDN switches perform a simple data plane functionality: packet forwarding. The control plane is performed by the logically centralized SDN controller (sometimes called the network operating system), which controls the SDN switches through an interface [9]. The SDN controller provides another layer of network abstraction upon which SDN applications can be built. When running HPC applications in an SDN based HPC system, the applications run on the compute nodes connected to the SDN switches, the applications use the services provided by the SDN controller to perform communications.

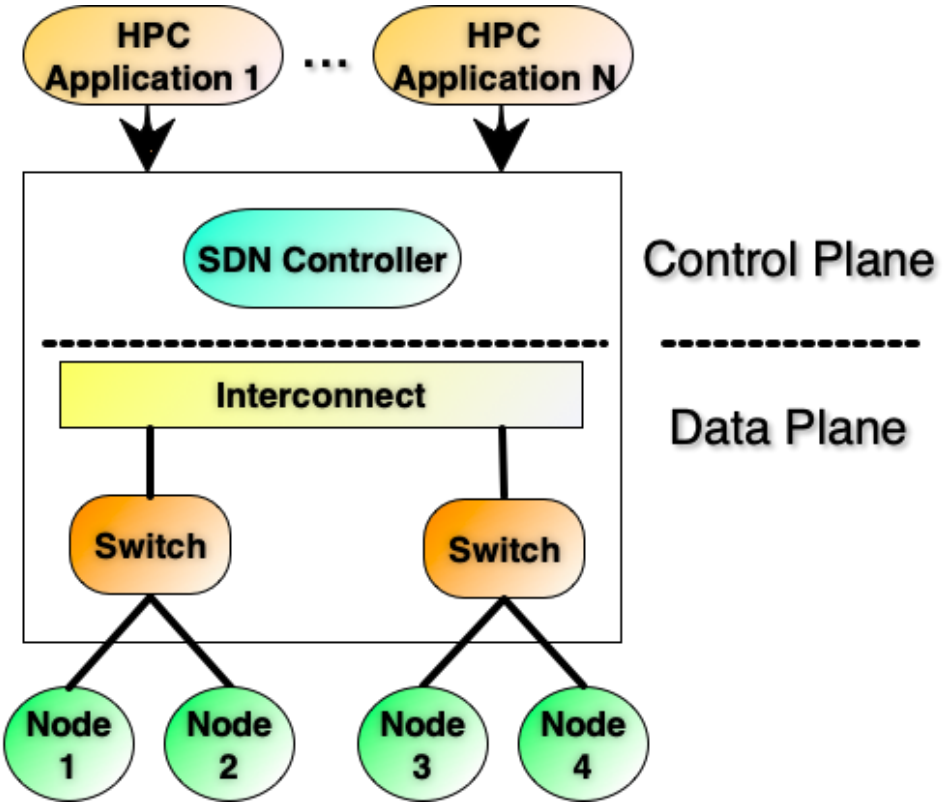

Figure 2.8: High-level overview of the SDN-based HPC System

2.5 Related Works

This dissertation focuses on two areas, incorporating Software Defined Networking (SDN) techniques to improve HPC application performance and optimizing hardware parameters for GPU-based HPC platforms.

In the first section, I analyze the integration of SDN techniques in HPC environments. My work evaluates how network flow classification, phase identification, and SDN-based routing compare with the traditional routing used in fat-tree topologies; it also tries to adapt these techniques for tapered fat-trees, aiming to determine whether incorporating SDN techniques helps run HPC applications efficiently on an HPC system. Although SDN is successful in other domains, it has not been fully explored in HPC. Studies by Faizian et al. have looked at SDN and adaptive routing in dragonfly topologies, providing a foundation for my work [17]. Since the introduction of SDN and OpenFlow [21], the technology has gained acceptance in industry and academia. Extensive research has explored SDN capabilities in the HPC domain. Arap investigated techniques for efficient MPI collective communications using SDN [6], while Takahashi evaluated the performance of the MPI_Allreduce operation on an SDN cluster [65]. Additionally, MPI_Reduce operations on SDN clusters have been developed, and efforts are underway to build new MPI libraries that take advantage of SDN capabilities. These studies show the potential of SDN to improve communication efficiency in HPC environments [48, 64].

In the second section, I explore the optimization of hardware parameters for next-generation GPU-based HPC platforms. Inspired by Jain et al.'s work on fat-tree configurations [32] and studies on dragonfly and jellyfish topologies by Rahman et al. [57] and Zaid et al. [4], I investigate how different hardware parameters affect network performance through system wide simulation. Simulations are crucial for evaluating HPC network performance. Previous research has utilized simulation tools like TraceR-CODES to analyze various system configurations. Jain and Bhatele, for example, used this simulator for detailed analyses of different systems, focusing on scalable topologies such as dragonfly, express mesh, and fat-tree [11]. These studies examined the performance of these topologies under different conditions, considering factors like the number of nodes, routers, and links, which are vital for understanding system costs and performance. They studied multi-job workloads with diverse communication characteristics to assess how link bandwidth, the number of rails, planes, and tapering influence system efficiency [32]. My research looks at the impact of varying the number of GPUs per node, network link bandwidth, and NIC scheduling

policies within fat-tree and dragonfly topologies. This aims to address how to strike a balance between computation and communication capacities by changing the hardware parameters as HPC systems evolve towards fewer, more powerful nodes.

In summary, my research optimizes HPC platforms by integrating advanced networking paradigms like SDN and bridges the gap between computational capacity and communication efficiency through network-parameter analysis.

CHAPTER 3

DESIGN AND EVALUATION OF TECHNIQUES FOR HPC PLATFORMS WITH SDN-CAPABLE INTERCONNECTS

Software-defined networking (SDN) [38] has shown great promise and has been widely deployed in data centers, campus networks, and wide-area networks. SDN has the ability to manage traffic at the flow level using the logically centralized global network view and to optimize the network resource utilization for global optimality, which may significantly improve network performance over the traditional networking infrastructure [8].

Although SDN features are also attractive to high performance computing (HPC) systems and applications, SDN has yet to be widely adopted in the HPC domain. Existing SDN techniques optimize for Internet and data-parallel applications (e.g. Hadoop and map-reduce applications) [28]. The communication characteristics of HPC applications are different from those of Internet and data-parallel applications. For example, many HPC applications simulate physical processes over numerous time steps, with each time step performing similar tasks: the execution of such applications exhibits phased behavior with alternating computation and communication phases. During the computation phases, few communications are performed; and the communications often repeat themselves in different time steps. Additionally, communications in HPC applications are often static in that they are known to the application developers or can be analyzed statically or dynamically [18, 29]. Exploring such features in HPC applications and systems will allow SDN to support communications more effectively and to perform its tasks more efficiently.

In this work, I develop techniques to adapt SDN to HPC workloads and systems, taking HPC application characteristics into account. The techniques include flow identification, phase identification, and flow scheduling. Flow identification identifies the types of flows in applications, which is essential for an SDN-capable network to achieve high performance. Phase identification identifies communication phases in applications, which allows network resources to be utilized more effectively. Flow scheduling schedules the communication to achieve target optimization objectives. To maximize the effectiveness, my techniques treat static and dynamic communications in HPC

applications differently. Dynamic communications are handled using techniques similar to those in the traditional SDN networks. For static communications, I propose to enhance SDN with an API for HPC applications to give hints to the SDN system (e.g. whether a flow is an elephant flow or will likely be an elephant flow). For such communications, the network system relies on the upper layer to obtain communication information. This is similar to the intent-based API [13] where application developer and the network system work together for flow and phase identification.

I conducted extensive simulation experiments using the TraceR-CODES [30, 47, 32] PDES [22] simulator on a 3-level fat-tree topology [39] [19]. My simulation results reveal that my techniques improve the performance over the existing SDN scheme for applications with both static and dynamic communications. The main contributions of this work include the following:

- I identify the features in HPC applications that can be used to enhance the effectiveness of SDN.
- I develop techniques to adapt SDN to HPC applications and systems by exploiting the identified HPC features.
- I perform extensive simulation to evaluate and validate the proposed techniques.

3.1 Flow Identification in HPC Environments

Data centers and HPC systems differ significantly in terms of their traffic characteristics. While the majority of traffic in a data center is small-scale, unpredictable, and not localized to any one level of the switch, traffic in an HPC application is primarily near-neighbor traffic, and may not be modest flows. Traditional flow identification methods may not be effective in HPC environments. To effectively support SDN in the HPC environment, novel techniques that take HPC communication characteristics into consideration are proposed. Since a significant portion of communications in HPC applications are static, I propose to have an API for applications to provide flow information to the network directly. For dynamic communications, I develop a machine learning based approach for flow identification.

Figure 3.1 shows the high-level view of the proposed flow classification systems. The flow classification techniques can be classified into two types, those *with no extra user information* and those *with user information*. The components below SDN API are similar to traditional SDN systems. The flow classifier may take traffic statistics from SDN switches and performs flow classification with no extra user information. The classifier also allows HPC applications (SDN

user) to directly give hints about their communications to the network through an API (similar to the intent-based API [13]), and classifies such flows based on user information.

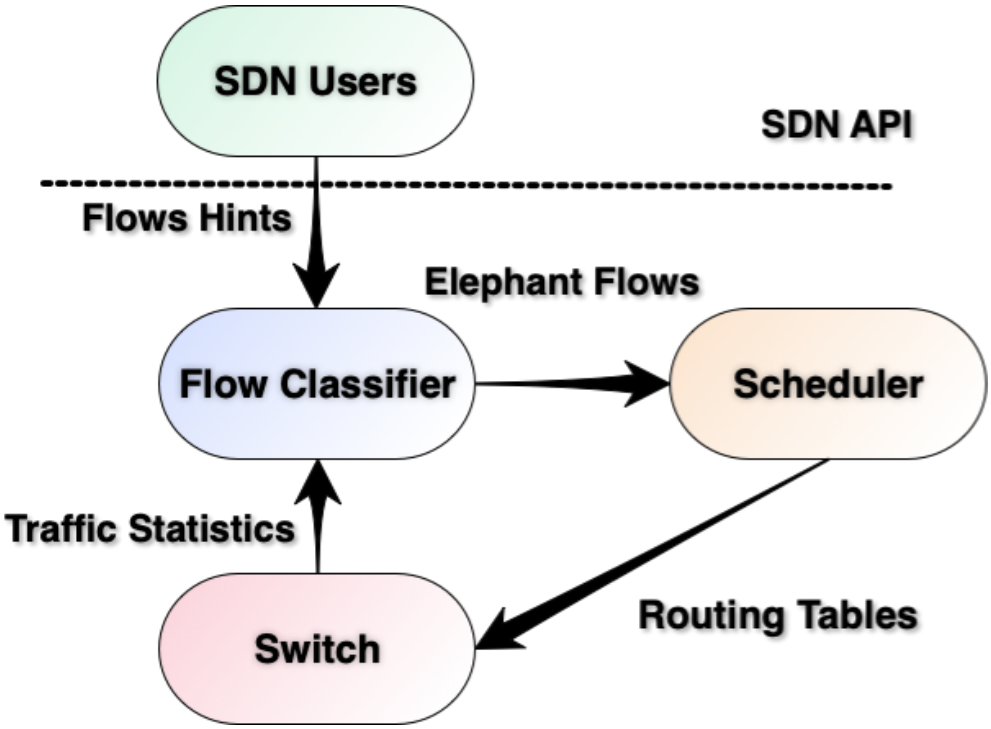

Figure 3.1: High-level view of flow classification schemes

3.1.1 Flow Identification Without User Input

Threshold-Based Scheme: Existing flow classification methods including end host-based management [67], packet sampling [62] [2], and polling per-flow statistics [68], do not assume the type of applications. In this work, I compare my proposed techniques with the classic elephant flow detection using a polling-based approach, which is based on Hedera's architecture [20]. Hedera's control loop comprises three main components: flow detection, channel calculation, and channel placement. Initially, significant flows are detected at the edge switches, and appropriate channels for these large flows are calculated using placement algorithms, taking into account their natural demand. These pathways are then placed on the switches.

Due to the overhead concerns, there is a limit how fast the flow statistics are gathered, and path calculation and installations are performed, which is typically in the order of seconds [20]. However, in the HPC environment, depending on applications, the time for each iteration may be much less than a second. Hence, these existing flow classification schemes will miss many iterations

of application execution and not effective for such HPC applications. To perform flow classification effectively for HPC systems, I develop a machine learning based approach using deep neural network (DNN) for flow classification.

Deep Learning-Based Scheme: As mentioned earlier, communications in HPC applications exhibits phase behavior. If such behavior can be characterized and learned, I can classify the flows quicker (after a small number of phases). The information to differentiate between large elephant flows from small mice flows is fundamentally captured by the time sequence of packets. By training a Deep Neural Network (DNN) model using the time sequence of packets from HPC workloads, the DNN model is able to recognize the patterns of elephant flows in HPC applications. I note that the patterns learned by the DNN model goes beyond simple statistics like the existing threshold-based scheme: the model can also reflect more sophisticated patterns such as the phased behavior.

Model Architecture: My DNN model consists of an input layer of size 300 to accept the input data which consists of data sent across various intervals of time and a single dense layer with one output neuron, which uses the sigmoid activation function to classify the flows as elephants or mice. The sigmoid function is commonly used for binary classification tasks as it outputs a value between 0 and 1, which can be interpreted as the probability of the input belonging to the positive class. The model uses the Nadam optimizer with a learning rate of 0.01. Nadam is an extension of the popular Adam optimizer that incorporates Nesterov momentum, which helps to accelerate convergence. The model is compiled with binary cross-entropy loss, which is a standard loss function used for binary classification tasks.

Data Collection and Model Training: I utilized the TraceR-CODES simulator to execute representative HPC workloads, namely Random-Permutation, Shift-256, Stencil3d, Stencil4d, Milc, Nekbone, Subcom3d-a2a, Kripke [41], Laghos, SW4lite [61], and AMG [56], for ranks ranging from 32 to 512. More detailed description of these workloads is given in Section 2.3. Flow statistics were collected every 0.3 seconds or 1% of the simulation to maintain minimal granularity. The flow data was gathered independently for each rank and merged to train the DNN model. For training purposes, the flows are marked as elephant or mouse flows with a cut-off of $3 * 10^8$ bytes of data transfer in 90 seconds: Elephant flows were those transferring more than 300 MB of data in 90 seconds, while mouse flows were those transferring less. I employed a min-max scalar to scale the flow data before inputting it into the dense neural network layer for forecasting. The developed model

was used offline to forecast network flows of elephants or mice for systems running a combination of the aforementioned applications.

During training, the model is trained on the input data and corresponding binary labels. The total data is split as follows 20% of the training data is used for validation, while the remaining 80% is used for training. Table 3.1 shows the model prediction accuracy: the model prediction accuracy is very high for the data. I first trained the model on the applications which I are going to use for my simulations and see how it performed which are Random-Permutation, Shift-256, Stencil3d, Stencil4d, Milc, Nekbone and later on I added Subcom3d-a2a, Kripke , Laghos, SW4lite, and AMG for training to see if the model is able to handle the new data, along with the existing data and still provide a accurate prediction. I do this to make sure that my model is capable of getting updated with new traffic as an when it comes.

Table 3.1: Accuracy of predicting elephants and mice flows by the DNN model for averaged across 32, 64, 128, 256 and 512 ranks

Applications	Accuracy
Random-permutation	100.0
Shift-256	100.0
Stencil3d	100.0
Stencil4d	100.0
Milc	100.0
Nekbone	100.0
AMG	100.0
Kripke	100.0
Laghos	100.0
Subcom3d	100.0
SW4lite	98.2

3.1.2 Flow Identification With User Input

For static communications in HPC applications like the ones in Stencil4d shown in Figure 2.5, the HPC application developer (or compiler and communication library) has the knowledge whether a communication is an elephant flow or not and can mark each flow in an MPI point-to-point communication as elephant flow or a mice flow or all flows in an MPI collective communication as elephant flows. With this approach, the SDN basically passes the flow identification task to the applications (done by application developer, compiler, or runtime library), which greatly simplifies the SDN operation.

There are different static communications in HPC applications. Consider MPI applications. The most complete information about a point-to-point communication includes the source MPI rank, the destination rank, and the message size. The communications in Stencil4d belong to this type. For such communications, users can give the hint when the application is loaded (when MPI ranks are mapped to physical nodes). For communications whose source and destination cannot be determined statically, but the size can be decided, the hints may be given at runtime when the communications are executed. The example of Laghos shown in Figure 2.6 belongs to this type. Such information can also be used for collective communications where a group of processes are involved in the communication. In the case when message size cannot be determined, the user may still use the API to indicate that a flow is an likely elephant flow with the knowledge of the applications. The SDN may use a simpler flow classification than the ones dealing with the most general unknown flows.

3.2 Phase Identification

HPC applications exhibit phased behavior with alternating computation and communication phases. Since communications in different phases do not overlap, network resources such as communication channels allocated to communications in one phase can be reused for communications in another phase. Hence, identifying communication phases, which is unique in the HPC environment, allows the SDN to manage resources more effectively.

3.2.1 Phase Identification With User Hints

Communication phases in an HPC application can be easily identified inside a program: there are often a section of code corresponding for the communications in the program. Figure 3.2 shows the code snippet for Laghos with communication phase marked. Here, after initiating MPI_Allreduce communications, the node performs MPI_Barrier to make sure all ranks have the data, before initiating the computation phase where it calculates the density. Now, the communication pattern may be different in the two communication phases in the given code as the comm world is calculated at runtime. Being able to reuse network resources in different phases will significantly improve network resource utilization. I propose to have an API for HPC application to give information about phases. With the API, the user can insert a marker before and after each communication phase in the program to inform the network the starting and ending of a communication phase. With the assistance from the user, the SDN network can detect communication

phases in an HPC application without minor overheads: once all processes for an application enter a computation phase, the application is in the computation phase and resources for communications in the previous phase can be released; as soon as one process enters a communication phase, the application enters the communication phase.

```
LagrangianHydroOperator (...){
        ...
        ParMesh *pm = H1FESpace.GetParMesh();
        MPI_Allreduce(&loc_area, &glob_area, 1, MPI_DOUBLE, MPI_SUM,
            pm->GetComm
());
        ...
}
```

Figure 3.2: Laghos code snippet

3.2.2 Dynamic Communication Phase Identification

Without the hints from application, communication phases in an HPC application can be detected by finding the computation period in the application when very few communications are performed. My dynamic communication phase identification algorithm is shown in Figure 1. I first decide the interval value for a minimal computation phase (e.g. 100ms). The phase detection algorithm is executed periodically at the interval boundary to determine whether phase is a communication phase or a computation phase depending whether the total data communicated in the application during the phase passes a threshold value. If a current phase is a computation phase and the previous phase is a communication phase, network resources allocated for the application in previous phases are released.

Algorithm 1: Dynamic phase identification algorithm

```
1 Let Total_data be all data sent by the application in the current interval
2 if (Total_data > Threshold) then
3     Current phase is a communication phase
4 else
5     Current phase is a computation phase
6     if (The previous phase is a communication phase) then
7         Release resources allocated in previous phases
```

3.3 SDN Routing

The SDN controller has the information of the traffic in the network. It then uses the traffic information and network information to schedule the traffic. I call the scheduling algorithm *SDN-based routing*. Clearly, SDN-based routing can be different for different network topologies and is constrained by the underlying network routing mechanism such as single-path routing or adaptive routing. In this work, I develop SDN-based routing for both full-bisection bandwidth fat-trees and tapered fat-trees, with support for single-path as well as adaptive routing.

I consider routing (scheduling) elephant flows that are long-lived and bandwidth-intensive. In the discussion, I will model the system as a directed graph $G = (V = S \cup N, E)$, where N is the set of compute nodes, S is the set of switches, and V is the set of switches (S) and compute nodes (N), and E is the set of directed links. The traffic demand is represented by a set of elephant flows with each flow being denoted as a source-destination pair $f_i = (s_i, d_i)$, where $s_i, d_i \in N$.

$$D = \{f_1 = (s_1, d_1), f_2 = (s_2, d_2), ..., f_n = (s_n, d_n)\}$$

Given a traffic demand D, the objective of an SDN-based routing scheme is to achieve load balancing. In the following, I will first present my SDN-based routing for networks with single path routing and then discuss the scheme for networks with adaptive routing.

3.3.1 Single Path Routing

For single-path routing, all packets of a flow follow the same path. For a traffic demand $D = \{f_1 = (s_1, d_1), f_2 = (s_2, d_2), ..., f_n = (s_n, d_n)\}$, I will denote the path for flow f_i to be p_i, which consists of a set of directed links. For a given routing R, let L_l be the set of flows that are routed through link l. The load of the link l is the number of flows using the link ($|S_l|$). The load of a path p is the maximum load of the loads on all links in the path: $L_p = \max_{l \in p} |S_l|$. L_p is referred to as the path load. The load of the network for traffic demand D is the maximum of loads among all links:

$$L_D = \max_{l \in E} |S_l|$$

The design objective of my SDN-based single-path routing schemes is to minimize L_D for traffic demand D. My first algorithm, *SDN-greedy*, is a heuristic algorithm while the second algorithm, *SDN-optimal*, is optimal for full-bisection fat trees.

SDN-greedy. *SDN-greedy* is a lightweight, single-path routing algorithm designed to operate on both full-bisection and tapered fat-tree topologies. As shown in Algorithm 2, the controller processes each elephant flow in order. For each flow, it enumerates all feasible paths between the source and the destination. For each candidate path, it computes the maximum link load for the path. The controller then selects the path with the smallest maximum link load and assigns path to the flow. By repeating this process for all elephant flows in the current scheduling phase, the algorithm constructs a routing table that seeks to keep the network-wide maximum link congestion low. This greedy strategy effectively spreads traffic across the network. However, this greedy algorithm may not result in the optimal scheduling for a traffic demand, especially when the traffic is dense.

Algorithm 2: SDN-greedy routing

Input: Elephant flows in a phase D
Output: Routing table *Routing_table* for all flows in D

```
1 Function SDN-greedy(D):
2     Initialize an empty map Routing_table
3     foreach f_i = (s_i, d_i) ∈ D do
4         Get current link loads in the network
5         Compute all possible paths for f_i
6         Compute the path load for each of the possible paths based on the current link
            loads of the network
7         Add the path with minimum path for f_i to Routing_table for f_i
8     return Routing_table
```

3.3.2 SDN-optimal

While *SDN-greedy* spreads the flows to paths with minimum path loads in a greedy manner, and does not guarantee to achieve optimal L_D for a given set of elephant flows, *SDN-optimal* considers the set of flows as a whole and achieves optimal L_D for full-bisection fat trees. In the following, I will first describe SDN-optimal for full-bisection fat trees. After that I will discuss how to extend it for tapered fat trees.

Given traffic demand $D = \{f_1 = (s_1, d_1), f_2 = (s_2, d_2), ..., f_n = (s_n, d_n)\}$, I define $SRC_{s \in N} = \{f_i = (s_i, d_i) | s_i = s\}$ and $DST_{d \in N} = \{f_i = (s_i, d_i) | d_i = s\}$. SRC_s is the set of flows in D whose source node is s and DST_d is the set of flows in D whose destination node is d. I define the node load for a given D as

$$NL_D = \max(\max_{s \in N}\{|SRC_s|\}, \max_{d \in N}\{|DST_d|\})$$

The node load for a traffic demand is either the maximum number of flows coming from the same source or the maximum number of flows going to the same destination in the traffic demand. In a fat tree topology, since each compute node connects to one link to a switch, I have the following theory.

Theorem 1: For a fat-tree topology and a given traffic demand D, for any single path routing scheme, $L_D \geq NL_D$.

Proof: Since each compute node only connects one out-going link and one incoming link. Any single path routing scheme must route the flows from a source to the out-going link that connects to it and route the flows to a destination to the in-coming link to the destination. Hence,

$$L_D = \max_{l \in E} |S_l| \geq \max_{l \text{ is a out-going link of } n \in N} |S_l|$$

and

$$L_D = \max_{l \in E} |S_l| \geq \max_{l \text{ is an incoming link of } n \in N} |S_l|.$$

Hence,

$$L_D = \max_{l \in E} |S_l| \geq \max(\max_{s \in N}\{|SRC_s|\}, \max_{d \in N}\{|DST_d|\}) = NL_D.$$

□

SDN-optimal consists of two steps. Given a traffic demand D, the first step is to partition D into NL_D permutations. In the second step, the algorithm uses a contention free scheduling algorithm to schedule each of the NL_D permutations. Since with a contention free scheduling algorithm, each permutation can be routed with no link contention in a full-bisection tree. Hence, for each permutation, the maximum link load among all links in the network is at most 1. As a result, scheduling NL_D permutations will have a maximum link load among all links in the network of NL_D. In other words, with *SDN-optimal*, $L_D \leq NL_D$. Hence, *SDN-optimal* is optimal for any traffic demand on a full bisection fat tree. The following theorem summarizes this discussion.

Theorem 2: For a full bisection fat tree and *SDN-optimal*, for any traffic demand D, $L_D = NL_D$: *SDN-optimal* is optimal for the full bisection fat tree. □

Next, I will give details of the two steps in the algorithm.

Step 1: Partitioning D into NL_D permutations.. Given the traffic demand D, the algorithm first constructs a bipartite graph $G = (S, D, E)$, where S and D are source and destination nodes in D, and each edge $e \in E$ represents a flow: if $(s, d) \in D$, then there is an edge from $s \in S$ to $d \in D$. This initial bipartite graph is then augmented to be a NL_D-regular multi-bipartite graph by

adding dummy nodes and edges. Multi-bipartite graph allows multiple edges between two nodes. I first add dummy nodes such that $|S|$ is the same as $|D|$. After that, I add dummy edges to make each node in S has a degree of NL_D and each node in D has a degree of NL_D. For each node in $|S|$, if its degree is less than NL_D, I find a node in D whose degree is less than NL_D and add a dummy edge between the two node. This process is repeated until all nodes have NL_D degree. Once the NL_D-regular multi-bipartite graph is build, the algorithm then applies edge-coloring (via Kőnig's Theorem) to decompose the graph into NL_D disjoint perfect matchings [37]. Each matching corresponds to a permutation $P_1, P_2, ..., P_{NL_D}$. The dummy edge is then removed from the obtained permutation to yield the final permutations. The algorithm is described in Algorithm 3

Algorithm 3: Flow partitioning into disjoint permutations

Input: Flow set D
Output: NL_D disjoint permutations $P_1, ..., P_{NL_D}$
1 Construct bipartite graph $G = (S, D, E)$ from flows in F
2 Pad G with dummy nodes and edges to make it NL_D-regular multi-bipartite graph
3 Apply edge-coloring to obtain NL_D disjoint matchings $P_1, ..., P_{NL_D}$
4 Remove dummy flows from each P_i
5 **return** $P_1, ..., P_k$

Step 2: Contention-free scheduling for each permutation. Finding a contention-free schedule for a permutation on a full-bisection fat-tree topology has been investigated for more than two decades, and a number of efficient algorithms are now available [53, 58, 69, 55]. In this work, I represent the permutation with a connection matrix,[53] and apply the coloured-matrix algorithm of Rodríguez *et al.* [58], thereby ensuring that each permutation is routed without link contention.

SDN-optimal for Tapered Fat-trees. In a tapered fat-tree, bandwidth reduction occurs at higher layers, such as the aggregate to core level has less links than leaf to aggregate level. This architectural tapering leads to insufficient link capacity when routing permutation traffic, where the number of flows often exceeds the number of available links at higher levels. As a result, *SDN-optimal* does not work on tapered fat-trees since achieving a contention-free assignment for all flows within a single permutation becomes infeasible.

To overcome this limitation, the SDN-optimal routing strategy partitions the full permutation into multiple sub-permutations. Each sub-permutation is constructed such that the number of flows passing through each layer of the network matches the number of available links at that layer. This ensures that within each sub-permutation, contention-free routing is still possible using the techniques previously employed.

My goal is to make sure that I create sub-permutations by selecting flows in such a way that the links in between each fat-tree layers have no contention and has the maximum utilization. To construct these sub-permutations, the algorithm first selects flows that traverse the core layer, typically inter-pod flows that consume both leaf-to-aggregate and aggregate-to-core links. It continues selecting such flows until all available core-level links are utilized. At this point, adding more inter-pod flows would introduce contention. The algorithm then fills the remaining capacity at the aggregation layer by selecting intra-pod flows, which consume only leaf-to-aggregate and aggregate-to-leaf links. The result is a sub-permutation that fully utilizes available link resources without exceeding capacity at any layer. This allows Hall's Marriage Theorem [12] [26] to be applied to guarantee a conflict-free routing for each sub-permutation.

Algorithm 4: Sub-permutation construction

Input: F_{core}: Set of flows that traverse the core switch
F_{agg}: Set of flows that only traverse the aggregate switch
c: Number of available core-to-aggregate links per sub-permutation
a: Number of available aggregate-to-leaf links per sub-permutation
Output: P_{list}: list of sub-permutations

```
P_list ← ∅;
while F_core ≠ ∅ or F_agg ≠ ∅ do
    P ← ∅;
    core_links ← c;
    agg_links ← a;
    // Stage 1: Add core-level flows
    foreach f ∈ F_core do
        Add f to P;
        core_links ← core_links − 1;
        Remove f from F_core;
        if core_links == 0 or F_core == ∅ then
            break
    // Stage 2: Add aggregate-level flows
    foreach f ∈ F_agg do
        Add f to P;
        agg_links ← agg_links − 1;
        Remove f from F_agg;
        if agg_links == 0 or F_agg == ∅ then
            break
    Append P to P_list;
return P_list;
```

3.3.3 Adaptive Routing

Adaptive routing enables traffic to be split across multiple paths adaptively based on the network condition, improving bandwidth utilization and often reducing congestion. This is a very effective routing scheme for fat-tree. However, in many scenarios where flow paths overlap significantly, causing contention. On the other hand, for a full bisection bandwidth fat-tree, any permutation traffic including the shift traffic pattern can be scheduled without contention using single-path routing. Hence, for such traffic pattern, using single path routing can achieve higher performance than adaptive routing. Based on this observation, I propose *SDN-adaptive* that adapt between these two strategies: if the traffic demand D is less than a permutation, use SDN-optimal; otherwise, use the underlying adaptive routing.

3.4 Performance Evaluation

The proposed techniques have been extensively studied using the TraceR-CODES simulator [30, 47]. TraceR-CODES is a software tool suite used for performance analysis of parallel and distributed applications, and it is specifically designed to simulate large-scale scientific applications running on high-performance computing systems. In the following, I will first discuss the experimental setup and the extensions that are added to TraceR-CODES to support the evaluation. After that, the performance results will be presented.

3.4.1 Experimental Setup

The experiments are performed on two distinct fat-tree configurations: a 1024 node full bisection fat-tree and a 1536 node 3-to-1 tapered fat-tree with a 3:1 tapering ratio. In both configurations, each switch is equipped with 32 ports. All 32 ports of the core and aggregate switches are fully utilized, connecting exclusively to other switches to preserve the hierarchical structure of the topology. The proposed SDN techniques for HPC environments including flow identification, phase identification, and SDN routing are added to TraceR-CODES. Flow identification schemes with and without user input described in Section 3.1, including the threshold-based scheme, the DNN-based scheme, user input based scheme are incorporated in TraceR-CODES. To support the machine learning based flow identification scheme, a trained DNN model has been integrated into TraceR-CODES using Google TensorFlow's C APIs. To ensure compatibility with the TensorFlow libraries, I utilized MVAPICH 2 as the MPI implementation and adopted C++14 from GNU version 9.0.1 as the

compiler standard. The DNN model is able to process real-time network statistics and dynamically predicts network flows during the simulation runtime. The user-input based scheme is supported by marking each MPI call with user hints. The functionality to support phase identification with and without user information is added to the simulator, and the simulator can be configured to use a particular phase identification scheme. For dynamic phase identification, if the total data sent across the entire network is less than 100 KB during a given interval, the network is classified as being in the computation phase. For routing, three types of SDN routing mechanisms are added: *SDN-greedy* that allocates paths by minimizing congestion in real time using simple heuristics, *SDN-optimal* that computes globally least-congested paths by evaluating all possible routes, and *SDN-adaptive* that dynamically adjusts routing decisions based on the current traffic patterns and flow types. Each of these methods is described in detail in earlier. To mitigate the impact of various types of delay on our data, specifically in the communication aspect of the experiment, we have configured the router delay, network interface controller (NIC) delay, software delay, and remote direct memory access (RDMA) delay to zero. This setup allows us to eliminate any extraneous delay factors and isolate the effects of the communication process on our data analysis. The Table 3.2 outlines the network configuration parameters used in the simulation.

Table 3.2: Network parameters for simulation of SHS

Parameter	Value
Packet Size	8192 Bytes
Switch Radix	32
Link Bandwidth	11.9 GB/s
Eager Limit	64000 Bytes
NIC Scheduler	Round-robin

3.4.2 Application and Workloads

Eight representative applications are used in the study: `Random permutation`, `Shift`, `Stencil3d`, `Stencil4d`, `Milc`, and `Nekbone`, `Random permutation mixed`, `Stencil mixed`, and `Random permutation third`. All of the applications except `Random permutation mixed`, `Stencil mixed`, and `Random permutation third` have been described in detail in Chapter 2.

- **Random permutation mixed**: This pattern alternates between two distinct random permutation communication phases, separated by a computation phase of 100ms. Each commu-

nication phase uses a different source-destination pair, introducing dynamic changes in the communication structure.

- **Stencil mixed**: This application combines a `Stencil3d` phase followed by a `Stencil4d` phase, with an intermediate computation phase of 100ms. The transition between patterns allows evaluation of the routing system's responsiveness to changes in spatial communication demands.
- **Random permutation third**: In 3-to-1 tapered fat-tree there are a three flows which contends for one uplink. To have a complete no contention scenario, we used this traffic pattern where we only selected one third of the flows that are originally present in a random-permutation traffic.

To ensure accurate analysis of communication computation transitions in the Random permutation mixed and Stencil mixed, I enforce explicit synchronization barriers between phases. This guarantees that each communication phase is fully completed before the next begins, preventing overlap and allowing isolated assessment of routing behavior across phases.

For the stencil applications, the problem dimensions are defined based on spatial decomposition across processing nodes. With 1024 nodes, the Stencil3D application is partitioned into a 3D grid of 8 nodes along the x-axis, 8 along the y-axis, and 16 along the z-axis. In the case of Stencil4D, the domain is decomposed into 8 nodes along the x-axis, 8 along the y-axis, 4 along the z-axis, and 4 along the w-axis, forming a four-dimensional near-neighbor structure. For 3-to-1 tapered fat-tree with 1536 nodes, the Stencil3D configuration extends to $8 \times 8 \times 24$, while Stencil4D uses $8 \times 8 \times 6 \times 4$, maintaining the same spatial layout logic.

All of these applications exhibit diverse communication behaviors ranging from synthetic near-neighbor patterns to production-level scientific codes and serve as appropriate test cases to assess how well SDN techniques manage different types of network traffic. The result of various SDN techniques which are used in HPC is presented here

3.4.3 Evaluation of flow classification techniques

In this section, I investigate how alternative flow-identification schemes affect the SDN routing algorithm across a suite of applications executed on two network topologies: a full-bisection fat-tree and a 3-to-1 tapered fat-tree. Application-level communication time serves as the primary performance metric. Throughout the experiments, I hold the routing policy constant at SDN-optimal and employ User-phase identification mechanism, systematically varying only the flow-identification technique.

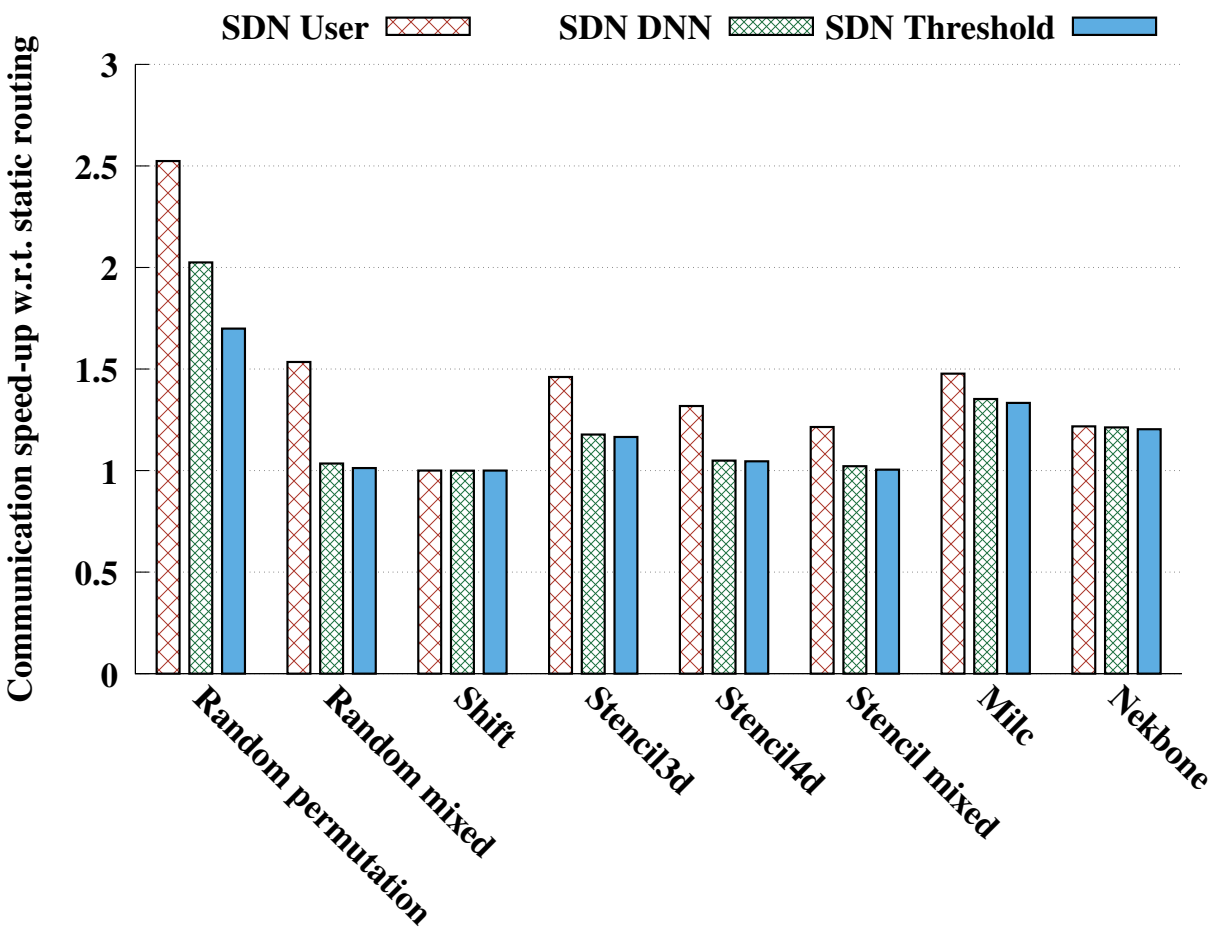

Figure 3.3: Comparison of flow detection in full bisection fat-tree of 1024 nodes

Performance Under full bisection fat-tree. Figure 3.3 presents the communication speed-ups I obtained with three flow-identification strategies, SDN User, SDN DNN, and SDN Threshold—relative to a static routing baseline. Any bar exceeding unity indicates that my method reduced communication time with respect to static routing. Across every workload, I observe that the SDN User scheme delivers the greatest acceleration because it tags elephant flows at their inception and reroutes them accordingly from the start of the simulation. In the random permutation kernel, for example, SDN User achieves a 2.52× speed-up, whereas SDN DNN and SDN Threshold reach only 2.02× and 1.69×, respectively. The same pattern holds for production codes: my SDN User configuration accelerates stencil 4d by 1.32×, MILC by 1.48×, and Nekbone by 1.22×, each time outperforming the threshold detector. SDN DNN consistently ranks second. Although it incurs a fixed 0.3 ms classification latency, I find that its data-driven model recognizes the repetitive communication patterns of HPC applications more rapidly than the polling-based threshold

approach. This advantage is most evident in random permutation, with more than a 20 percent improvement over the threshold method, and it persists in real applications such as Nekbone, where SDN DNN attains a speed-up of 1.40 compared with 1.30 for the traditional threshold-based detector.Taken collectively, my results indicate that augmenting SDN routing with user-phase detection and a lightweight DNN classifier confers a statistically significant performance advantage over static routing. Moreover, this hybrid scheme matches or exceeds the communication efficiency of the conventional threshold-based detector while achieving faster flow-classification latency.

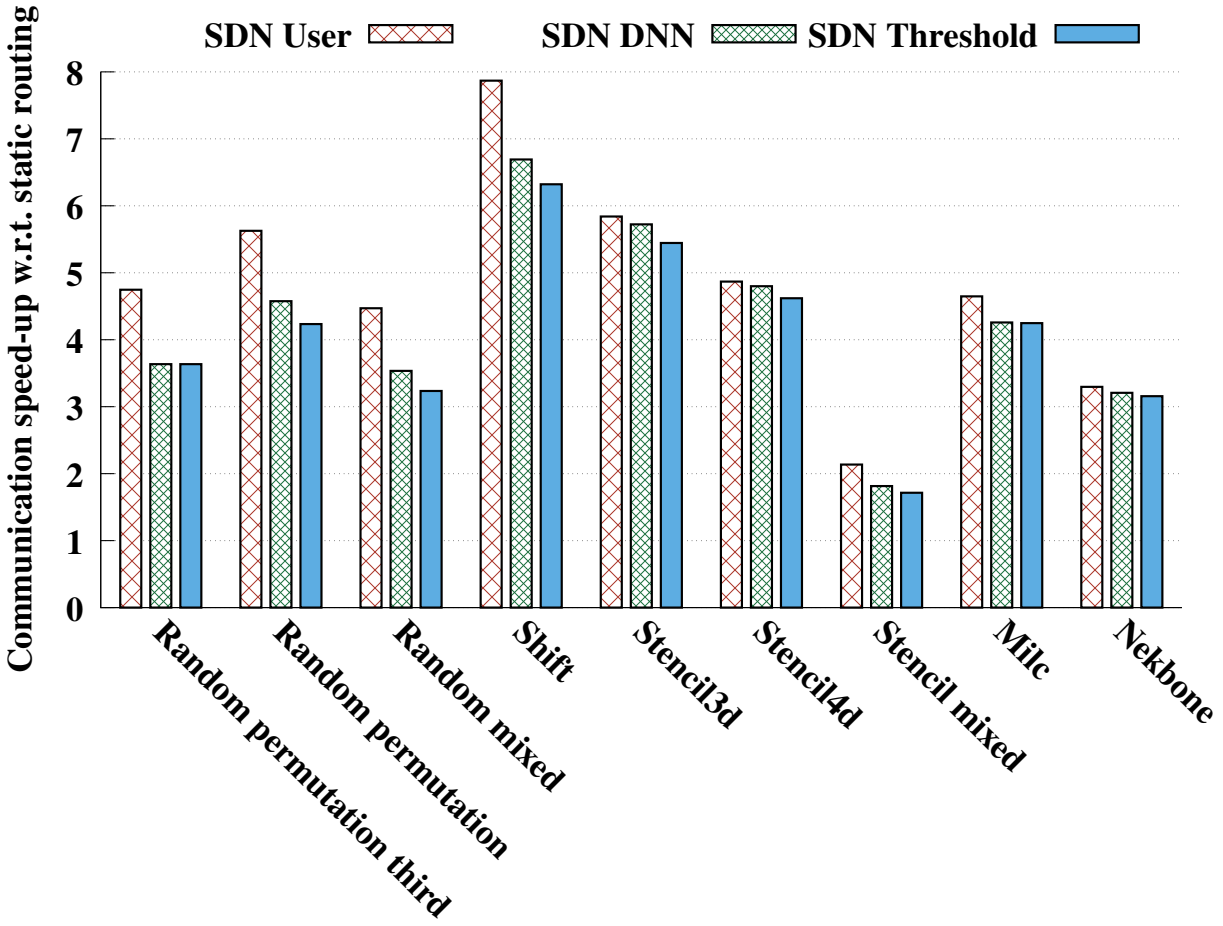

Figure 3.4: Comparison of flow detection in 3-to-1 taper fat-tree of 1536 nodes

Performance under 3-to-1 tapered fat-tree. The Figure 3.4 illustrates the communication speedup achieved by different flow detection techniques (SDN User, SDN DNN, and SDN Threshold) under a 3-to-1 taper fat-tree topology with 1536 nodes and a 3:1 tapering ratio. This topology emphasizes the impact of reduced bandwidth at higher levels of the fat-tree structure. In

a 3-to-1 tapered fat-tree topology, even a random permutation pattern represents a relatively dense communication workload. To better demonstrate the effectiveness of my technique, I introduced Random permutation third by removing two-thirds of the communication from a random permutation of 1536 nodes, retaining only eight out of the 24 communication flows passing through a leaf router. My evaluation revealed that SDN User consistently achieved the highest speedup, on an average 4 times faster compared to static routing, highlighting the benefits of early user-provided flow identification that enables optimal traffic balancing from the start. In contrast, SDN DNN and SDN Threshold exhibited moderate speedups of approximately 1.5x and 1.2x, respectively, due to delayed flow detection. In real applications such as Milc and Nekbone, the performance gap between techniques narrowed, though SDN User remained the top performer.

3.4.4 Evaluation of Phase Identification

This section examines how different phase-identification schemes perform across multiple applications on both a full-bisection fat-tree and a 3-to-1 tapered fat-tree. Throughout the tests I fixed routing to SDN-optimal and activated the SDN-user flow detector, changing only the phase-identification method.

Performance under full bisection fat-tree. Figure 3.5 presents the communication speed-ups I obtained with three phase-identification strategies, User-phase, Dynamic-phase, and No-phase, relative to a static-routing baseline. Any bar exceeding unity indicates that phase awareness allowed the network to shorten communication time. I focus on the random mixed and stencil mixed kernels, where I deliberately inserted computation intervals to expose the benefits of phase detection. Across both workloads the User-phase scheme delivers the largest acceleration, because the application signals each phase boundary and the controller can release bandwidth immediately. In random mixed, User-phase attains a 1.53× speed-up, whereas Dynamic-phase and No-phase reach only 1.48× and 1.40×, respectively. A similar pattern emerges in stencil-mixed where the User-phase achieves 1.21×, Dynamic-phase 1.20×, and No-phase 1.11×. Dynamic-phase consistently ranks second as it has a delay of 100ms to detect a phase change. Although it incurs that delay, its on-line inference still recognizes the shift from communication to computation which helps is releasing the resources quickly.

Performance under 3-to-1 tapered fat-tree. Figure 3.6 presents the communication speed-ups with different phase identification techniques in 1536-node, 3-to-1 tapered fat-tree. Here,

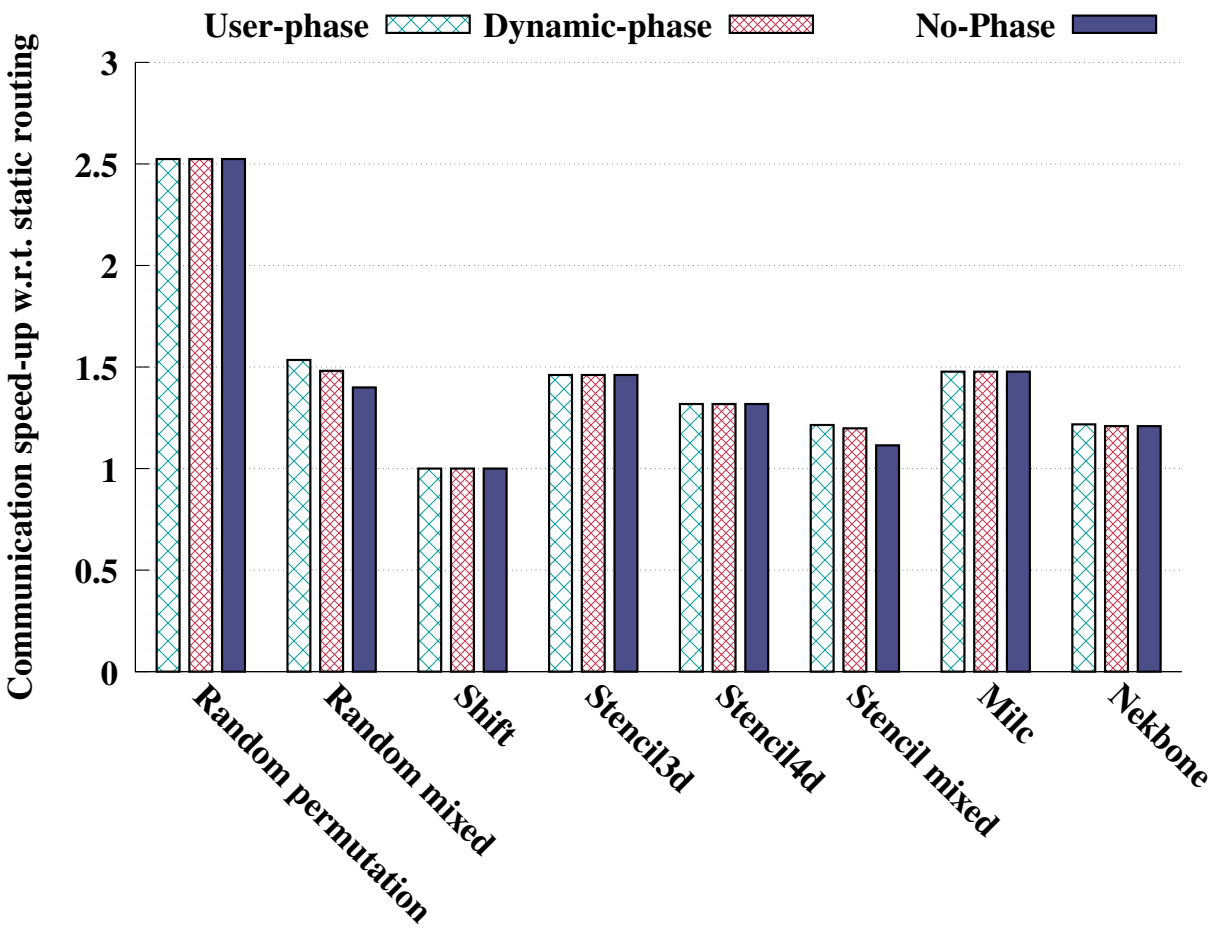

Figure 3.5: Comparison of phase identification in full bisection fat-tree of 1024 nodes

in random-mixed, User-phase achieves a 4.53× speed-up, whereas dynamic-phase and no-phase reach 4.42× and 4.18×, respectively; thus, explicit phase cues deliver an additional 8% benefit over dynamic phase detector that waits for 100 ms to infer phase changes. A similar pattern appears in stencil-mixed, where user-phase attains 2.21×, dynamic-phase 2.15×, and no-phase 2.04×.

Taken together, these data indicate that phase-based optimisation combined with SDN routing is especially valuable in fat-tree topologies, where user supplied phase information yields the greatest benefit, a lightweight DNN detector closes most of the gap while avoiding additional efforts from user to pre-determine the phases, and both approaches outperform the case where no phase detection is performed.

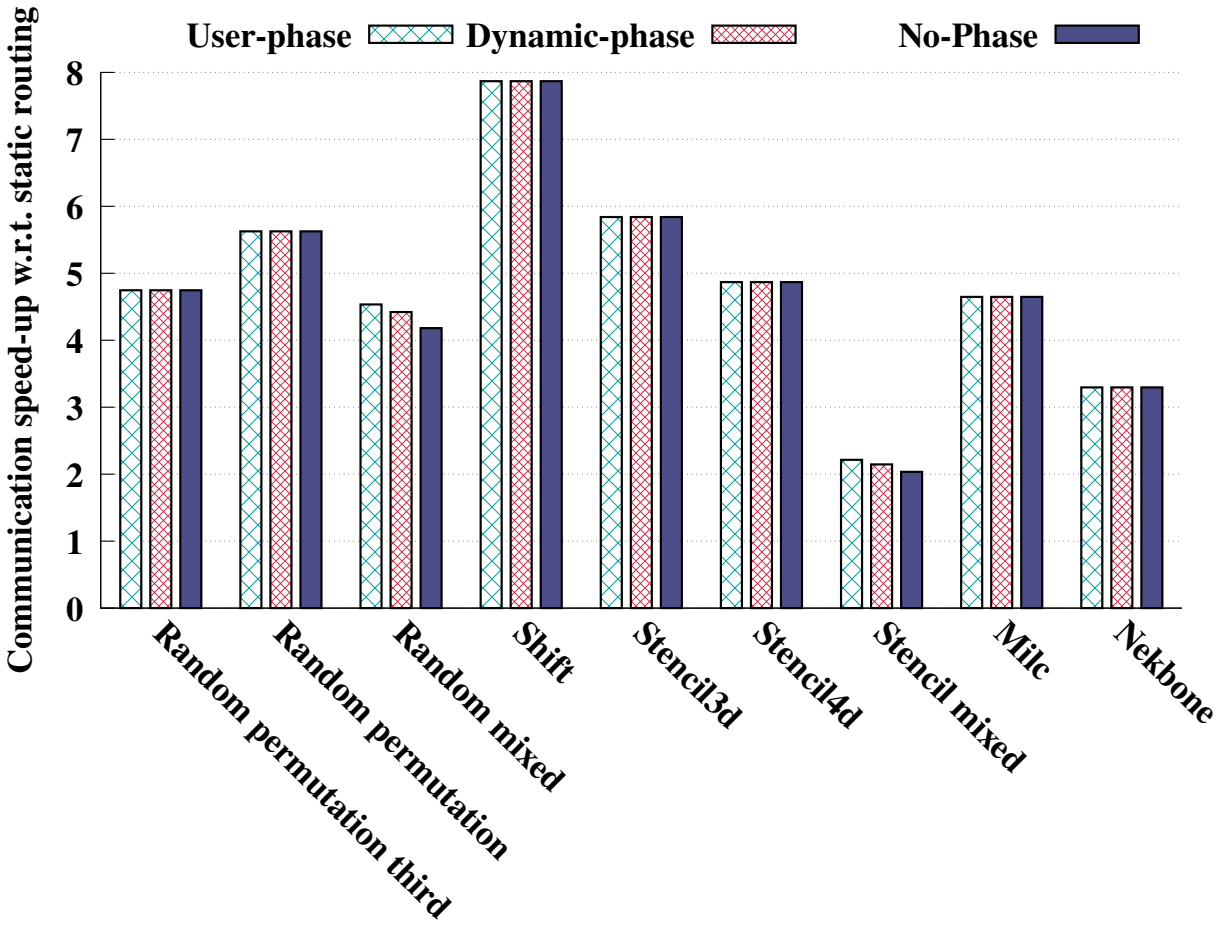

Figure 3.6: Comparison of phase identification in full bisection fat-tree of 1536 nodes

3.4.5 Evaluation of SDN-based Routing

In this section, I assess the efficacy of several SDN routing strategies over two network topologies, a full-bisection fat-tree and a 3-to-1 tapered fat-tree. To isolate the influence of the routing algorithm, I keep the phase identification scheme as User-phase and the flow detection scheme as SDN User, varying only the routing method itself across the experiments.

Performance under full bisection fat-tree. Figure 3.7 makes clear that software-defined routing delivers a substantial and consistent benefit over static routing. All three SDN routing schemes, SDN-greedy, SDN-optimal, and SDN-adaptive has a communication speed-up well above 1, meaning they always shorten application communication time compared to single path static routing without SDN features. The effect is most pronounced in the random permutation workload, where SDN-optimal reaches a 2.52× speed-up (a 152 2.10× speed-up in random permutation. Real

applications like Milc and Nekbone shows the same pattern. In the four-dimensional stencil4d kernel SDN-optimal achieves 1.32×, and in the computation-heavy MILC application it still provides a 1.48× gain. These results indicate that simply enabling an SDN controller so that every path is chosen centrally and made contention-free can more than double communication performance. Adaptive multipath routing, the current state-of-the-art multipath alternative, occasionally edges ahead of SDN in very dense traffic (for example, a 1.97× boost in MILC), yet it also falls below the static baseline in the shift benchmark, where a single permutation when made to share paths leads to some congestion. For this kind of situations where the traffic density is low or moderate, a hybrid SDN-adaptive mode recovers that loss by defaulting to the optimal SDN paths at start-up. Taken together, the data shows that SDN routing consistently surpasses static routing and in many cases exceeds adaptive routing, establishing centralized, contention-free path selection as a powerful and practical tool for accelerating HPC communication.

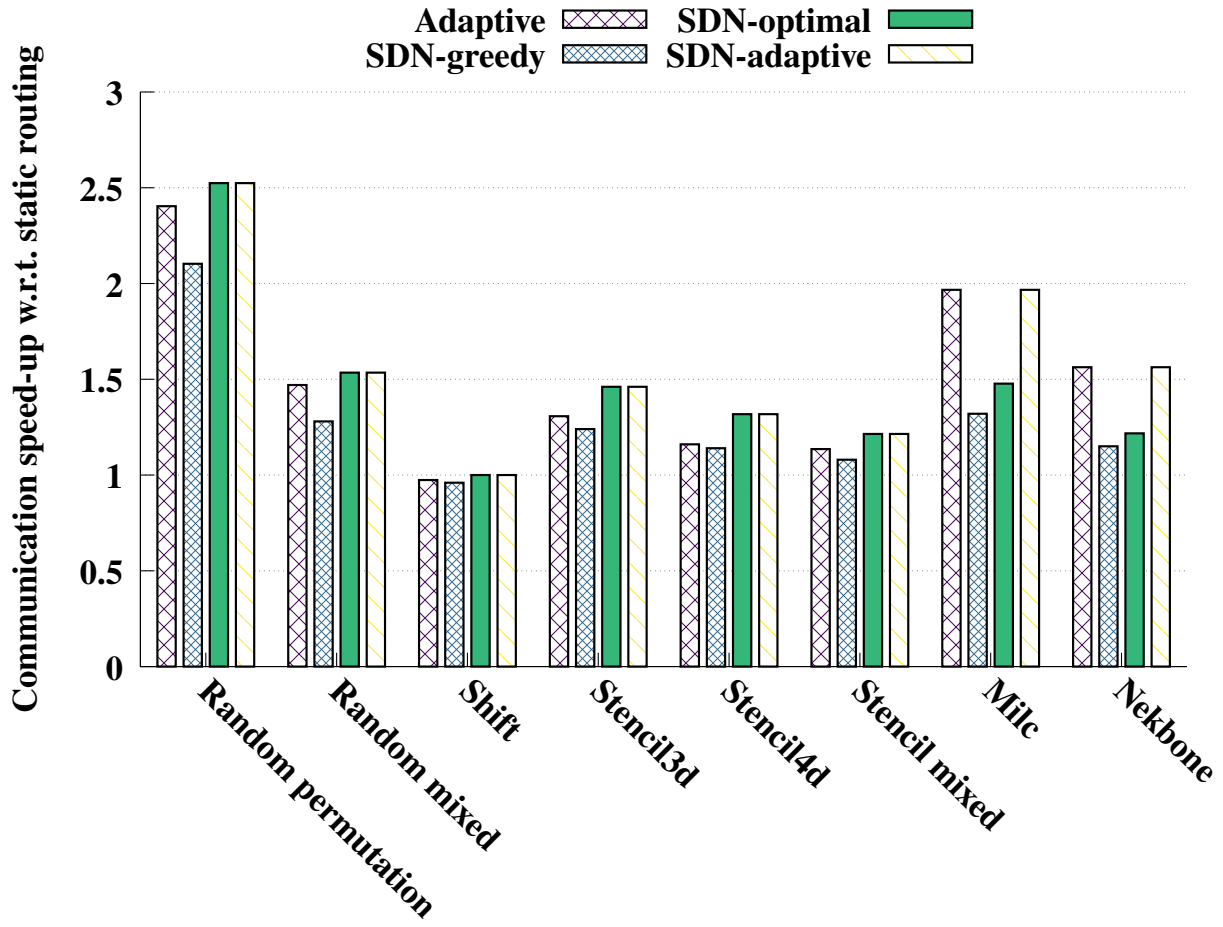

Figure 3.7: Comparison of routing techniques in full fat-tree of 1024 nodes

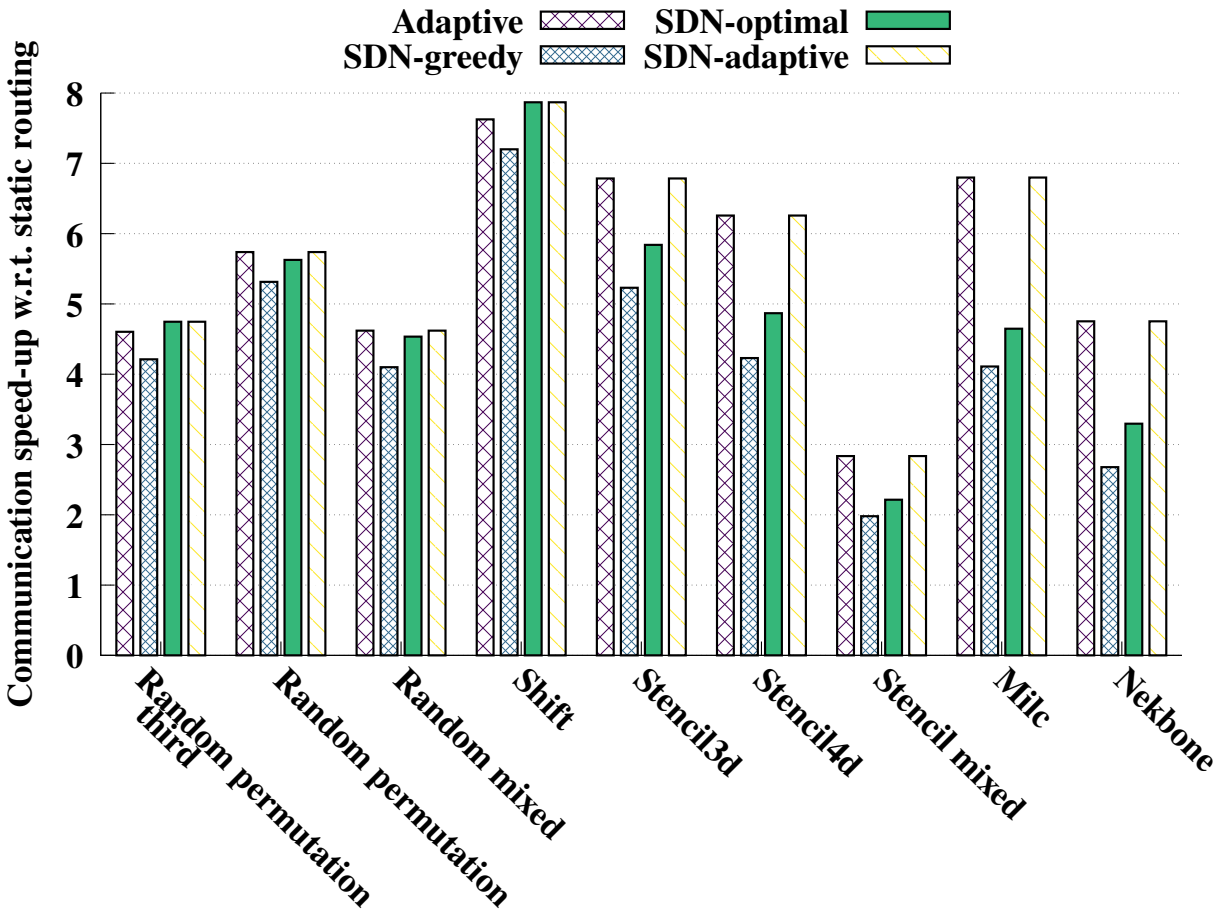

Figure 3.8: Comparison of routing techniques in 3-to-1 taper fat-tree of 1536 nodes

Performance under 3-to-1 tapered fat-tree. Figure 3.8 makes clear that SDN routing gives very large gains over static routing on the 1536-node, 3-to-1 tapered fat-tree. The SDN-optimal scheme is best overall, cutting communication time by factors of 7.87× in shift, 5.63× in random permutation, and 4.75× in random permutation third. Even the simpler SDN-greedy still reaches 5.31×, 4.21×, and 4.12× in the same three workloads. Tapering removes two-thirds of the upward bandwidth, so a full random permutation already counts as a moderately dense workload: many flows share the same core links, and multipath adaptive can spread them slightly better than single-path SDN-optimal (5.74× versus 5.63×). When I thin the traffic to random-permutation-third, each core link carries only one flow; under this simpler pattern SDN-optimal's centrally chosen, contention-free paths move ahead of adaptive (4.75× versus 4.60×). The same reversal appears in shift (7.87× versus 7.63×). In the very dense Milc and stencil3d kernels, however, adaptive keeps a small lead because its many paths can still distribute heavy load more evenly. In

summary, all SDN variants improve communication speed-up by roughly four- to eight-fold compared with static routing, and the hybrid SDN-adaptive mode matches SDN-optimal or adaptive routing in every case by switching to the globally computed paths or adaptive routing at start-up. These results show that, in a tapered fat-tree where congestion is stronger, centrally coordinated SDN routing can work better than the traditional static or adaptive routing widely used in fat-tree interconnect based HPC systems.

3.5 Summary

This chapter investigated the adaptation of SDN techniques to HPC systems whose communication patterns are distinguished by repetitive compute-and-communicate phases. Central to the study was an extensive TraceR-CODES simulation that examined three interlocking SDN control-plane mechanisms—flow identification, phase identification, and flow scheduling—across two representative interconnects: a 1024-node full-bisection fat-tree and a 1536-node 3-to-1 tapered fat-tree. The study first demonstrated that early, user-based flow classification delivers the greatest performance gains, while a deep-neural-network classifier offers an accurate, low-latency alternative and a threshold scheme provides a baseline. It then revealed that distinguishing computation from communication phases allows the network to quickly release resources, reducing worst-link congestion by in both full fat-trees and tapered counterparts. Finally, the evaluation of three SDN-based routing algorithms, SDN-greedy, SDN-optimal, and a hybrid SDN-adaptive strategy—showed that SDN-optimal minimizes maximal link load and outperforms conventional adaptive routing when traffic density is low, whereas SDN-adaptive dynamically shifts between optimal and adaptive behavior, achieving up to six to seven fold reductions in communication time under bandwidth-constrained conditions, specifically in tapered fat-trees. Collectively, these findings advance a holistic perspective in which rapid flow classification, phase-aware resource reclamation, and traffic-aware routing are orchestrated to elevate network efficiency. Because the proposed techniques depend only on flow semantics and relative link abundance, they extend naturally to the irregular, tapered fat-tree variants prevalent in production clusters, thereby offering a practicable pathway toward more agile and capable SDN-based HPC system. Drawing on the insights established in this groundwork, the following chapter identifies the hardware parameters that best support HPC applications in modern HPC systems.

CHAPTER 4

A SIMULATION STUDY OF HARDWARE PARAMETERS FOR FUTURE GPU-BASED HPC PLATFORMS

In this work, I delve into the issue of optimizing hardware parameters for next-generation GPU-based High Performance Computing (HPC) platforms, specifically addressing the challenges and opportunities presented by integrating multiple GPUs within compute nodes. This is motivated by the evolving landscape of HPC platforms, where there is a marked shift toward increasing computational capacity per node while concurrently reducing the overall number of nodes or endpoints in the system. Prior studies, such as those by Jain et al. [3], have laid the groundwork by evaluating the performance impact of various fat-tree configurations, offering valuable insights into network architecture's role in HPC systems. Similarly, research on topology and routing methods for Dragonfly networks by Rahman et al. [27] and on Jellyfish topologies by Zaid et al. [28] has advanced my understanding of network performance under different configurations. These studies, while foundational, primarily focus on network topologies and configurations rather than the integrated approach of combining only hardware parameters.

To provide context for this investigation, the increasing significance of compute acceleration devices, such as GPUs, which have drastically altered the computational landscape of HPC platforms. These devices have enabled a substantial increase in the computational capabilities of individual nodes, which is observed in the comparison between Sequoia at LLNL and Summit at ORNL. This transformation warrants a reconsideration of hardware architectural parameters to ensure that future HPC platforms can achieve optimal performance levels. This study aims to address the imbalance between computation and communication capacities that arises as there is an ongoing transition to HPC systems with fewer, more powerful nodes. The integration of multiple GPUs per node introduces new complexities in maintaining an efficient computation-to-communication ratio, making it imperative to explore hardware configurations that can mitigate these challenges. To tackle these issues, this study utilizes the TraceR-CODES simulation tool to analyze the impact of various hardware design parameters on the performance of realistic HPC

workloads. This investigation centers on three critical hardware parameters: the number of GPUs per node, interconnection network (topology, link bandwidth, etc), and network interface controller (NIC) scheduling policies. These parameters are evaluated within the context of two widely-used network topologies: fat-tree and dragonfly. The main conclusions of this study underscore the nuanced relationship between hardware parameters and system performance. The results show that the optimal configuration of GPUs per node, interconnection network, and message scheduling strategies significantly depends on the specific demands of the applications running on the HPC platform. For instance, communication-intensive applications may require higher network bandwidth to maintain performance levels as the number of GPUs per node increases. Conversely, computation-heavy applications may see minimal impact from changes in network bandwidth but could be affected by NIC scheduling strategies.

4.1 System Parameters

Network topology: In this study's experiments, the impact of the hardware design parameters are studied in the context of two widely used interconnect topologies: fat-tree and 1D dragonfly.

(1) 1D Dragonfly – 1D Dragonfly [34] is a two-level direct network topology: switches form groups with a fully connected intra-group topology and groups are connected with an inter-group topology. The topology has three important parameters [34]: the number of compute nodes in each switch (p), the number of links in each switch that connect to other switches in the same group (a), the number of links in each switch that connect to other groups (h). A balanced dragonfly in general requires $a = 2p = 2h$. In this study the parameters are set to $p = h = 8$ and $a = 16$. Each group has 16 switches and 128 compute nodes. The global link connectivity between groups follows the per-router arrangement [5]. The routing algorithm used is the progressive adaptive routing (PAR) [34, 5].

(2) Fat-tree – The other topology is a 3-level full bisection bandwidth fat-tree. In a 3-level full bisection bandwidth fat-tree, there are three types of switches: 1) core switches which are at the top layer to connect pods, 2) aggregate switches, which connect the leaf switches and form a pod, and 3) the leaf switches, which are connected to the compute nodes. In a 3-level full bisection bandwidth fat-tree, the number of uplinks in the aggregate and leaf switches is the same as the number of downlinks. For our study, the 3-level fat-tree is built using 32 radix switches. Each

leaf switch connects to 16 compute nodes and 16 aggregate switches. Each pod has 16 aggregate switches, 16 edge switches, and 256 compute nodes.

Number of GPUs per node: In this study, the number of GPUs in each compute node varies from 1 to 8 to analyze the impact of the increased computation density and the reduction of network endpoints on the system performance. Each GPU is assigned to one MPI processes; to simulate different number of GPUs per node, multiple MPI process are assigned to a node. The GPUs inside a node are connected in an all-to-all connection topology resembling the intra node connectivity of the Sierra system with NVlink. The bandwidth between GPUs within a node is set to be twice the network link bandwidth, so that it replicates that of Sierra supercomputer. The default setting for GPUs per node is 1 GPU per node. This is the default GPU per node setting whose performance is used to normalize other results.

In the experiments, when I increase the number of GPUs per node, I proportionally reduce the number of network endpoints, i.e. I make sure that for all network configurations, the total GPU count, as well the total MPI processes, is 2048. This is done to ensure that I compare systems that are of computationally equal capability as is often the case in the real world. Secondly, I make sure that each workload covers the entire network and no node is left empty during the simulation. Table 4.1 summarizes the network sizes used for each GPU per node setting, with the default setting being that of 1 GPU per node.

Table 4.1: Network sizes for different GPUs per node.

GPUs per node	1D Dragonfly	Fat-tree
1	16 Groups	8 Pods
2	8 Groups	4 Pods
4	4 Groups	2 Pods
8	2 Groups	1 Pods

Network link bandwidth: We set our baseline link bandwidth as x=11.9 GB/s, which is the peak achieved link bandwidth on Mellanox EDR networks such as the Quartz supercomputer at LLNL. To analyze the sensitivity of various compute capability equivalent systems to communication capability, I vary the bandwidth from x/16 (16 times slow down of the baseline) to 16x (16 times speedup of the baseline). In the rest of the document, I will use x to represent the base bandwidth, and will denote the network speed as x/16, x/8, x/4, x/2, x, 2x, 4x, 8x, and 16x.

Message scheduling: As the computation and communication density on the compute node increases, message scheduling performed by the NIC may have an impact on communication performance. In particular, scheduling schemes that alleviate head-of-line blocking may have significant benefits, especially when the link bandwidth is very high. In addition to head-of-line blocking, which is often mitigated by the use of virtual channels, message scheduling also affects congestion management and network utilization. Scheduling schemes that expose packets from multiple communicating-pairs to the network may perform better as it provides the network with the flexibility to use multiple network paths concurrently. To investigate the effect of message scheduling on a system with different network and different node compute capability, I compare the performance of FCFS, RR, and RR-N with different values of N on systems with different configurations.

FCFS scheduling: Messages are inserted at the back of the scheduling queue, as and when they arrive. During the packetization process, the scheduler keeps creating packets from the top of the queue until the entire message is packetized before it packetizes the next message in the queue.

RR scheduling: Messages are inserted into the scheduling queue of the network interface, as and when they arrive. During the packetization process, the scheduler creates one packet for a message and then moves to the next message: all messages are considered in a round-robin manner. RR not only allows concurrent communication progress for several communicating-pairs, but may also help the network in better utilizing multiple communication paths. While desirable, such a scheme is difficult to implement in the hardware as the number of concurrent messages can be very large.

RR-N scheduling: In this scheme, N is a parameter. RR-N is similar to RR, except that instead of packetizing every message in the scheduling queue in a round-robin manner, the scheduler packetizes the top N messages in the scheduling queue. For example, in RR-2, the scheduler only packetizes the first 2 messages for communication. This newly added scheme simulates the real world scenario where a limited number of hardware queues are available at a NIC, which are used to keep multiple messages in-flight concurrently.

4.2 Application and Workloads

I selected six applications of different computation and communication characteristics to create realistic HPC workloads. The applications include two communication-heavy kernels, Stencil4d[10] and Subcomm3d[10], two compute-intensive applications, Kripke[41] and Laghos[44], and two ap-

plications with a balanced communication-to-computation ratio, AMG[56] and SW4lite[61] (see Figure 4.1). The traces used in the study were collected using Score-P [36] on Vulcan, a Blue Gene/Q installation and Quartz, an Intel Xeon cluster at Lawrence Livermore National Laboratory (LLNL). The traces contain information about all MPI events executed on each MPI process, along with their timestamps. In addition, they also record user annotations such as loop begin and end for the main compute loop. A brief description for the applications in provided in Chapter 2.

Figure 4.1 presents the fraction of total execution time these applications spend in communication and computation when running with 32 processes. Computation is denoted by the red color, and non-overlapped communication is shown in green. At 32 processes, Stencil4d and Subcomm3d are dominated by communication. The communication-computation ratios were tuned in Stencil4d and Subcomm3d such that they replicate the runtime profiles of representative communication-intensive applications. Kripke and Laghos are dominated by computation with both spending more than 95% of the time in computation. AMG and SW4lite spend ∼80% of their time in computation and the rest in communication. Suitable computation scaling factors are used to alter the behavior of these traces to emulate running the computation on GPUs. Figure 4.1 shows how the computation-to-communication ratios change as these scaling factors are applied. Stencil4d and Subcomm3d spend most of their time in communication after compute scaling and the other applications now spend between 25-65% in communication.

The workloads in the study are created using the six HPC applications mentioned above at different process counts – 32, 64, 128, 256, and 512. In this study, the system supports up to 2048 processes. Thus, the sum of process counts in each of the workloads is exactly 2048. Each workload is obtained by iteratively randomly selecting an application and a job size until the total workload size has reached 2048. As a result, each workload has many jobs of different sizes, resembling the capacity workload of supercomputing centers [30]. This study's experiments use 20 such random workloads. To ensure that the reported performance of each job size of each application is representative, each job size of each application appears at least four times in the 20 workloads. This warrants that each job size of each application has been executed under different conditions in the experiments.

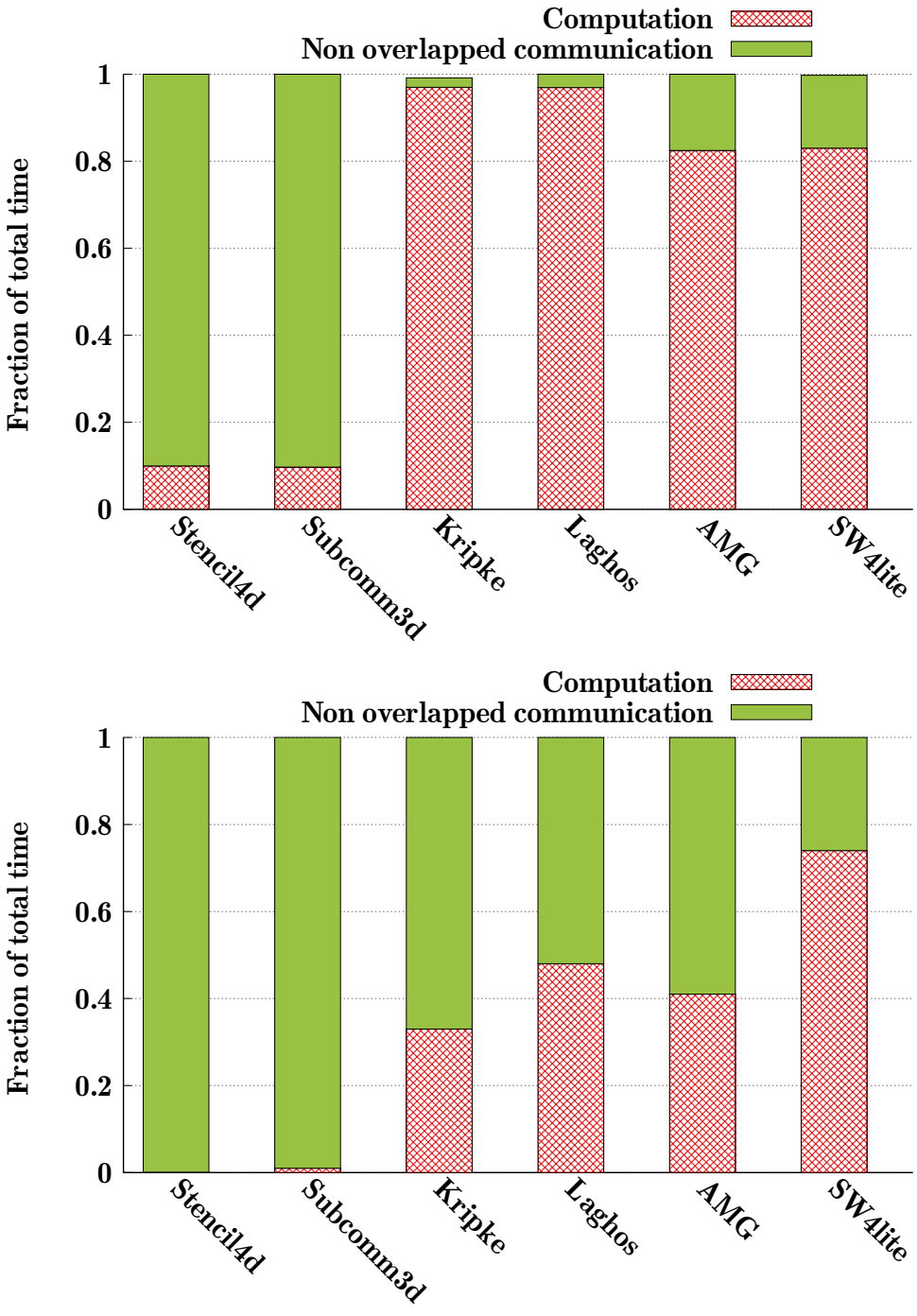

Figure 4.1: Computation and communication characteristics of all applications without scaling (left) and with scaling for GPUs (right) running on 32 processes.

4.3 Validation of Tracer-CODES

TraceR-CODES has been previously validated with micro-benchmarks and stand alone applications including pF3D, 3D Stencil, ping-pong, all-to-all, etc [31]. These validation studies were done for fat-tree networks and it was found that TraceR-CODES predicts the absolute value as well as the trends in the execution time with less than 15% error [31, 1]. However, these validation studies have been done with single job simulations. Further, these studies did not validate cross-platform and cross-network projections, i.e. traces were collected and projections were done for the same system.

To gain confidence in TraceR-CODES' prediction for cross-platform and cross-network multi-job workloads as well as in the new additions to TraceR-CODES, this work validates TraceR-CODES with three random multi-job workloads. The validation is done by 1) randomly creating three workloads that consist of representative HPC benchmarks with different communication and computation characteristics, 2) running the workloads on the Quartz supercomputer [42] at LLNL, 3) simulating the workloads using TraceR-CODES with the system parameters set to the values for Quartz, and 4) comparing the predicted job execution times from the simulations with the measured times on Quartz.

The three workloads are formed by selecting jobs from two communication intensive benchmarks (Stencil4d and Subcomm3d) and two computation intensive applications (Kripke and Laghos).

In this study, three workloads were run in a dedicated access time (DAT) on Quartz at LLNL, during this period no other jobs ran on the machine. It used linear mapping of job ranks to nodes and measured the execution time of each job in the workloads. For simulation with the TraceR-CODES framework, it used the exact system settings as Quartz: (1) create the exact fat-tree topology as Quartz using the arbitrary graph model; (2) set the values of the network parameters to the corresponding values on Quartz: 11.9 GB/s peak link bandwidth, 8 packets buffer size, 4096 bytes packet size, and so on; and (3) the jobs and processes in each workload are mapped to compute nodes exactly in the same way as they ran on Quartz.

The traces for driving the simulation were collected on Vulcan [45], a 5D-torus based Blue Gene/Q system. Since the computational capabilities of Vulcan are different from Quartz, the relative compute scaling factor between Vulcan and Quartz is calculated, and the computation regions of simulations were scaled accordingly. This setup helps to evaluate the projections when

the network (5D-torus vs fat-tree) as well as computational capability (IBM PowerPC vs Intel Xeon) of the traced system are different from the target system.

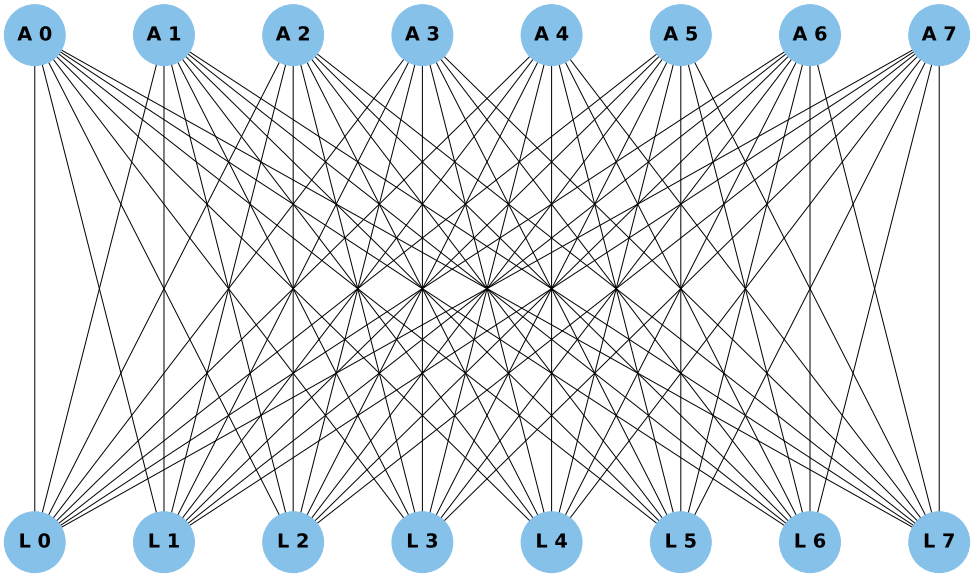

Figure 4.2: A Quartz pod with eight aggregate and eight leaf switches, and all links.

Quartz Topology: The Quartz system deploys a 3-level fat-tree, with a 2:1 tapering at each of its 84 leaf switches. There are 84 aggregate switches and 32 core switches. Each switch has a radix of 48 and each leaf level switch is connected to 32 compute nodes. Note that some ports in the aggregate switches and core switches are left unused. The 84 leaf level switches are divided among 11 pods. Figure 4.2 shows a Quartz supercomputer pod. Each pod consists of 8 leaf switches and 8 aggregate switches, which are connected in an all-to-all bipartite graph. Each arc drawn here represents two physical links. In contrast, a standard 2:1 tapered fat-tree would have 16 leaf switches in each pod, which are connected to 16 aggregate switches using one physical link each. We give these details of the Quartz topology to highlight that Quartz' fat-tree is different from the standard, symmetric fat-tree topology, as are the networks in most production systems. These differences are the main driver for the development of the *arbitrary graph model*.

Figure 4.3 shows the results of the validation. The horizontal axis, have each application and their corresponding job size used in various workloads. Each blue dot represents the average of the error percentage between the predicted runtime and the measured runtime for various instances of the given application-job size pair that appear across the three workloads. For example, since Subcomm3d jobs with a process count of 128 appears two times across the three workloads, their

average error percentage is computed to be -7.88%. It can be observed, that for all cases except 32-ranks Stencil4d, the prediction error is within 20%; and for all except 3 cases (32-rank Stencil4d, 32-ranks Kripke, and 64-ranks Kripke), the error is within 15%. These results suggest that TraceR-CODES predictions reasonably approximate the actual runtime on real systems for multi-job workloads even when the computational capability and underlying network are different.

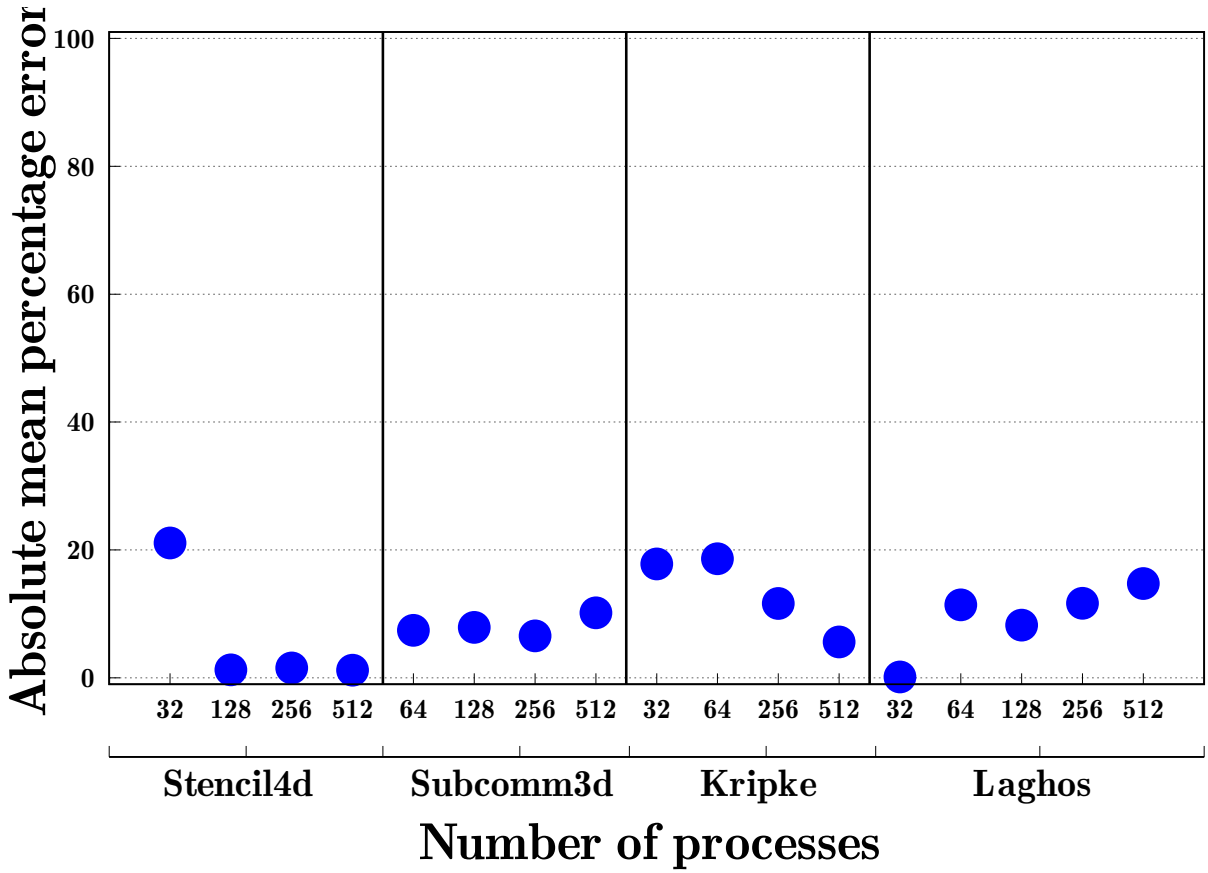

Figure 4.3: Validation of TraceR-CODES (mean percentage error in predicted runtime compared to the actual runtime).

4.4 Performance Evaluation

The results of simulation studies of various different architectural parameters which are prerepresented in Section 4.1 are shown below.

4.4.1 Impact of the Number of GPUs per Node

The number of GPUs per node determines the balance of computation to communication capacity of a system and thus is an important configuration choice in GPU-based HPC clusters. The impact of this parameter on different types of HPC applications is studied.

Figure 4.4 presents the relative performance of the applications running on fat-tree based systems with different number of GPUs per node. The speedup in the figure is computed relative to the performance when running with 1 GPU per node. For example, in Figure 4.4, Stencil4d with 32 processes has a speedup of 0.71 in the 2 GPUs per node mode. This implies that the performance of Stencil4d with 2 GPUs per node is 71% of the Stencil4d performance with 1 GPU per node. Other configuration parameters are held constant at their default values (1x network link bandwidth, FCFS message scheduling, etc.) The reported performance is the average across all occurrences of an application and a given job size in the 20 workloads. Note that across the different GPUs per node configurations, each application and job size combination gets exactly the same computing resources. More GPUs per node does not imply more computing power for a given application and job size combination; it simply implies that the computing resources are available in a more condensed manner on fewer, more powerful nodes.

In Figure 4.4, it is observed that for communication-heavy applications (Stencil4d and Subcomm3d), as the number of GPUs per node increases, application performance drops for most job sizes. This is because as more GPUs are placed per node, the effective communication resources available for each GPU reduce. However, the performance drop is not linear w.r.t. the effective communication resources because the mapping of multiple MPI processes to node results in some of the data being communicated within node. This data can make use of the high-bandwidth intra-node GPU links.

An opposite effect is observed in the simulations of the 1D dragonfly topology in Figure 4.5. In some cases, such as Subcomm3d on 32 and 64 nodes, a significant amount of traffic is converted to intra-node when using 8 GPUs/node, which results in performance improvement of the application. Another factor that impacts performance is that when all processes in a job are mapped to a single switch, the job is less susceptible to inter-job network interference than when the processes in a job are mapped to multiple switches in the interconnect. With 4 GPUs per node, a 32-process job is mapped to 8 nodes and a 64-process job is mapped to 16 nodes. With 8 GPUs per node, a 32-process job is mapped to 4 nodes and a 64-process job is mapped to 8 nodes. Each switch

in the fat-tree connects to 16 nodes and each switch in 1D dragonfly connects to 8 nodes: there are chances for the 32-process and 64-process jobs to be mapped completely within one switch and achieve higher performance.

For the next two applications (Kripke and Laghos), a noticeably different impact of changing the balance of communication to computation capability is observed. In the case of Kripke, more GPUs per node do not impact its performance. This is because the overall communication volume is low, and GPUs are often waiting on other GPUs to finish their computation. For Laghos, a slowdown primarily with 8 GPUs per node is observed. This indicates that having these many GPUs per node shifts the communication-computation balance and also the performance characteristics of the application.

Finally, for the last two applications (AMG and SW4lite), a gradual slowdown when more GPUs are incorporated per node, on both network topologies is observed. While this performance drop is not as high as the communication-heavy applications, it is noticeable for the 4 and 8 GPUs per node configurations. It is also found that for most applications that are sensitive to network performance, several factors including the communication pattern of the application, job mapping, and inter-job interference impact the execution time. For example, AMG and Laghos, experience higher slowdown in 8 GPUs per node configuration in workloads in which they are placed adjacent to communication-heavy applications. The typical reason for this slowdown is that communication-sensitive applications when mapped adjacent to similar applications contend for network resources, thus impacting the performance.

***Overall Observation**: Most applications run slower with four or more GPUs per network endpoint.*

In the experiments, all but one application (Kripke, which is not sensitive to network capabilities) slow down noticeably with four or more GPUs per network endpoint. Although part of the communication volume may be restricted to intra-node communication with more GPUs per node, this benefit is typically overshadowed by performance loss due to the reduction of the node communication to computation ratio.

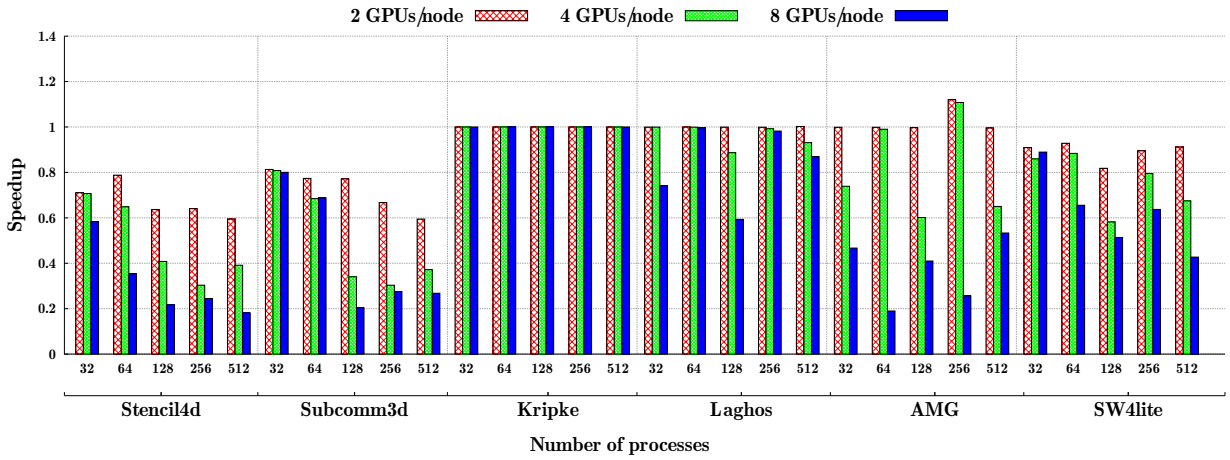

Figure 4.4: Speedup on fat-tree for various numbers of GPUs per node settings with respect to 1 GPU/node configuration.

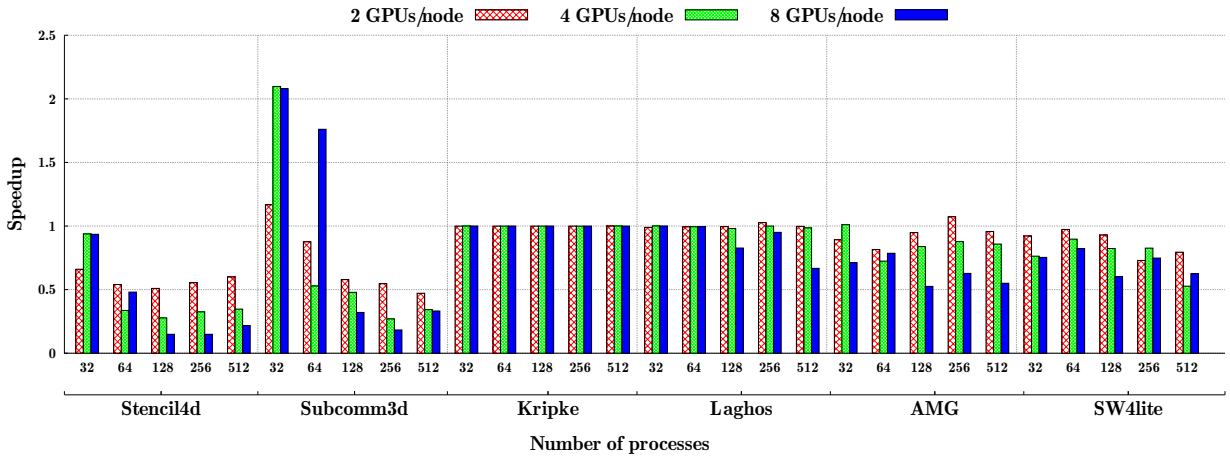

Figure 4.5: Speedup on 1D dragonfly for various numbers of GPUs per node settings with respect to 1 GPU/node configuration.

4.4.2 Impact of Network Bandwidth

In the previous section, as the number of GPUs per node increases, the default 1x network bandwidth becomes a performance bottleneck for many cases is seen. Thus, the impact of varying network bandwidth along with number of GPUs/nodes on application performance is studied next.

In the simulation experiments, it is observed that the impact of network bandwidth on jobs of different sizes shares similar trends. Hence, only the data for a job size of 128 processes is presented. Figure 4.6 shows the performance for the 4 GPUs per node configuration with varying

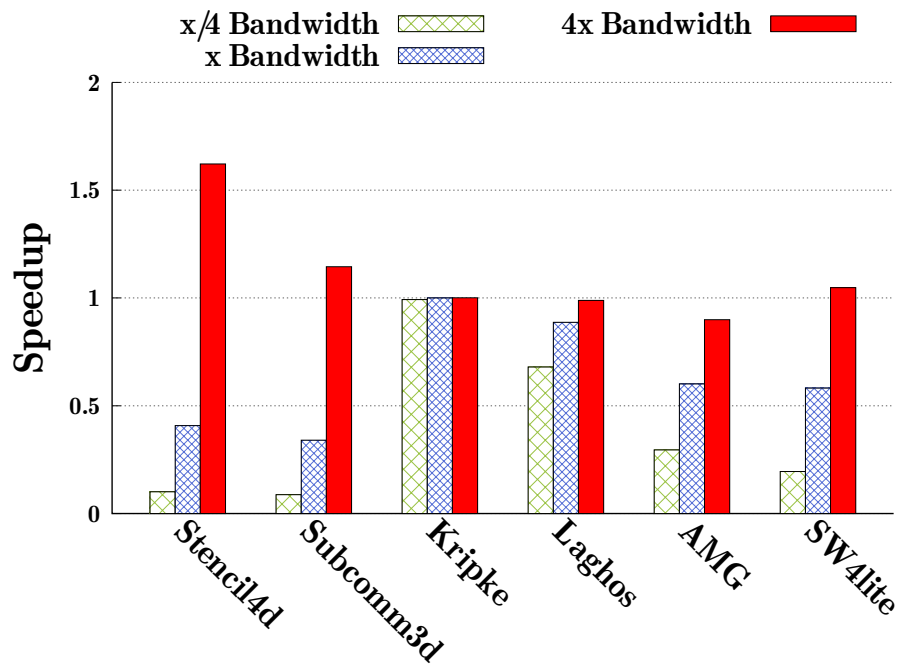

Figure 4.6: Speedup for the 4 GPUs/node configuration over 1 GPU/node in fat-tree, 1x network bandwidth configuration. Data is shown only for job sizes of 128 GPUs.

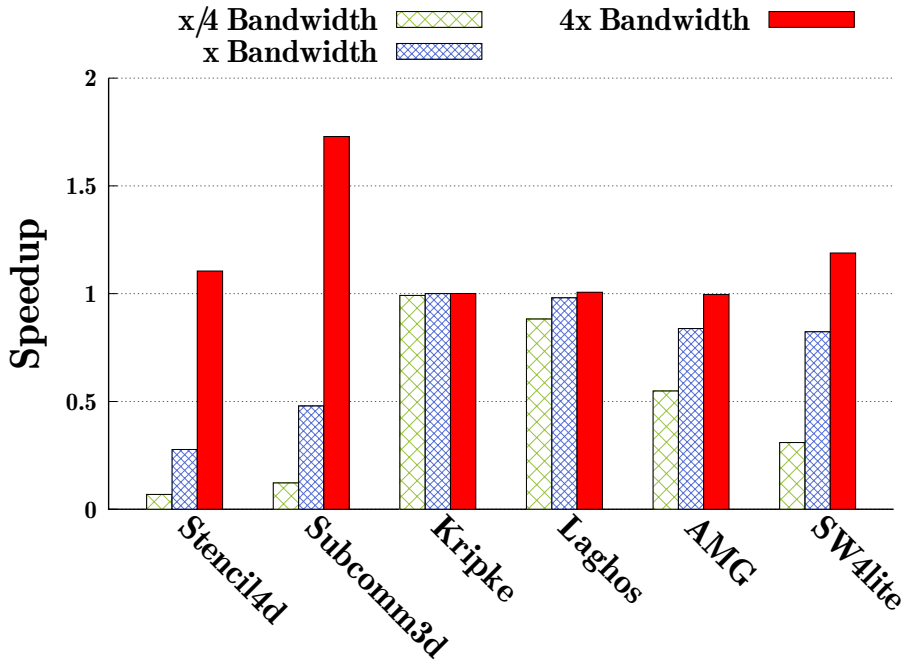

Figure 4.7: Speedup for the 4 GPUs/node configuration over 1 GPU/node in 1D dragonfly, 1x network bandwidth configuration. Data is shown only for job sizes of 128 GPUs.

network bandwidth relative to 1 GPU per node, 1x network bandwidth configuration on fat-tree network. We find that the network bandwidth has a significant impact on most applications. The gains are highest for communication-heavy applications such as Stencil4d and Subcomm3d. Conversely, the impact of reducing the network bandwidth is also highest for those. A similar trend is observed for the 1D dragonfly topology as shown in Figure 4.7.

Table 4.2 presents the minimum bandwidth required for each application and a given job size to achieve 90% of the performance of the default setting for the fat-tree topology. As expected, different types of applications have different bandwidth requirements. In general, communication-intensive applications require larger bandwidth to sustain the increased number of GPUs per node while computation-intensive applications have less bandwidth requirement. For example, for the 8 GPUs per node case with 512 processes job size, Stencil4d needs 8x network bandwidth to achieve 90% of the performance from the default setting; AMG and SW4lite need more than 4x bandwidth while Kripke only needs x/8 bandwidth.

Table 4.2: Minimum bandwidth required to achieve 90% of the performance of the default 1 GPU/node configuration for fat-tree

Applications	32 processes		512 processes	
	4 GPUs/node	8 GPUs/node	4 GPUs/node	8 GPUs/node
Stencil4d	1x	1x	4x	8x
Subcomm3d	x/2	x/2	4x	4x
Kripke	x/16	x/16	x/8	x/8
Laghos	x/2	2x	x	2x
AMG	4x	8x	4x	8x
SW4lite	2x	2x	2x	4x

Further, application requirements are also affected by the job size and the placement with other jobs. For example, 32-process Laghos ran slower in some workloads when mapped in the 8 GPUs per node configuration, which is why here double bandwidth is needed to get more than 90% speedup. It is also seen that sometimes communication-intensive applications such as Stencil4d and Subcomm3d require less bandwidth in 8 GPUs per node configuration than 4 GPUs per node configuration to reach 90% of the performance for 32 processes and 64 processes. This is mainly due to the fact that, with a larger number of GPUs per node, a significant

fraction of the communication happens within the same node. This indicates that future GPU-based platforms must consider their workloads to decide important networking hardware parameters. The results for 1D dragonfly, show a similar trend as that in fat-tree.

***Overall Observation**: Bandwidth requirement to sustain high performance depends on GPU density and job sizes.*

Our results show that each type of application has a sweet-spot for them to perform effectively. Hence, the design of a future GPU cluster should take its applications into consideration in order to achieve the maximum performance-cost ratio.

4.4.3 Impact of Message Scheduling in the NIC

The impact of message scheduling on system performance has not received sufficient attention in the community. To my knowledge, this is the first time that the impact of message scheduling on system and application performance is being studied systematically. Similar to the impact of the number of GPUs per node and network link bandwidth, the impact of message scheduling is similar for both fat-tree and 1D dragonfly. Thus, results for the 1D dragonfly are only discussed in detail.

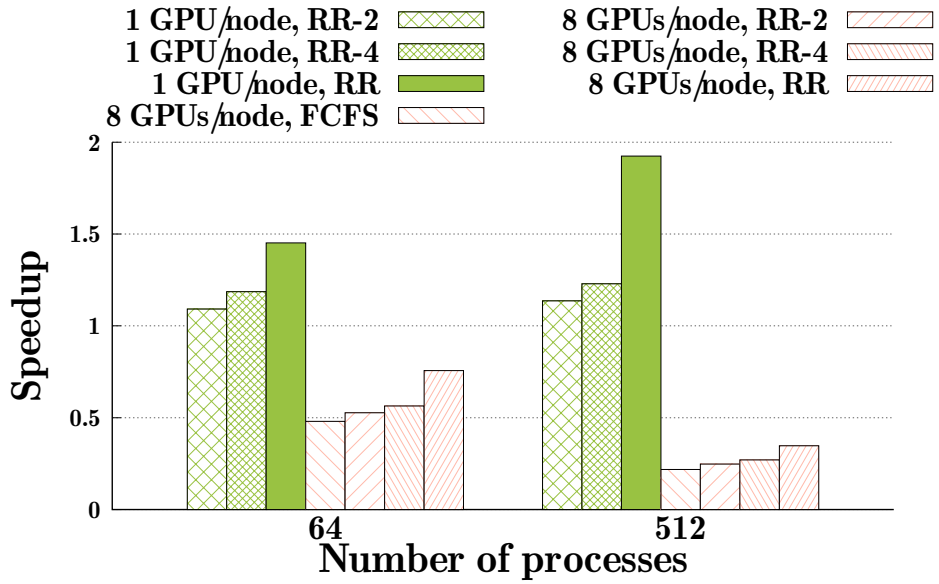

Figure 4.8: Results for Stencil4d (64 processes and 512 processes on 1D dragonfly)

Figure 4.8 shows the speedup for 64 and 512 processes (GPUs) of Stencil4d relative to the default case with 1 GPU per node and FCFS scheduling (network bandwidth is fixed at 1x for all

configurations). For the 1 GPU per node cases, the scheduling significantly affects the performance: the larger the number of messages the scheduler considers for packetization concurrently, the higher the performance. The RR scheduler reaches a speed-up of 1.45 for the 64-process job and 1.93 for the 512-process job in comparison to the default FCFS scheduler. A similar trend is observed for the 8 GPUs per node cases: the RR scheduler improves the speed up from 0.48 with the FCFS scheduler to 0.76 for the 64-process job, and from 0.22 to 0.35 for the 512-process job.

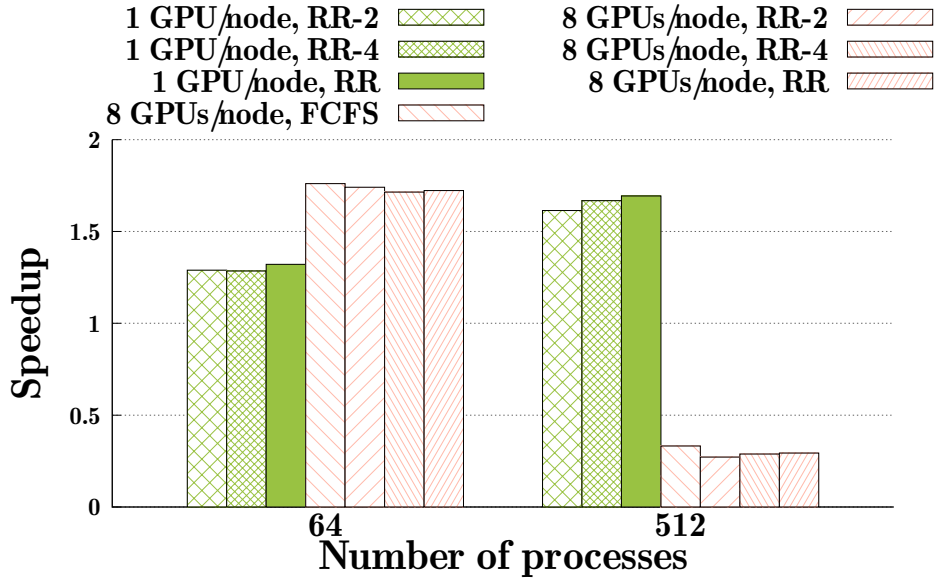

Figure 4.9: Results for Subcomm3d (64 processes and 512 processes on 1D dragonfly)

Figure 4.9 shows the speedup for 64 and 512 processes of Subcomm3d. For 1 GPU per node, RR scheduler performs better than FCFS. However, RR is only slightly better than RR-2 and RR-4 and achieves a 1.3 speed-up for the 64-process job and 1.7 speed-up for the 512-process job over FCFS. For 8 GPUs per node cases, all schedulers have similar performance with FCFS being slightly better than other scheduling schemes. Although both Stencil4d and Subcomm3d are communication-intensive, the impact of message scheduling is different. This is because the communication characteristics in these two applications are different.

Message scheduling has no impact on Kripke as Kripke is not sensitive to communication as seen earlier. Figure 4.10 shows the speedup for 64-process and 512-process simulations of Laghos. For 1 GPU per node cases, all schedulers have the same performance. For 8 GPUs per node cases, all schedulers have the same performance for the 64-process job, but RR has a significantly better performance than others for the 512-process job. As shown in Figure 4.5, for 512 processes (and

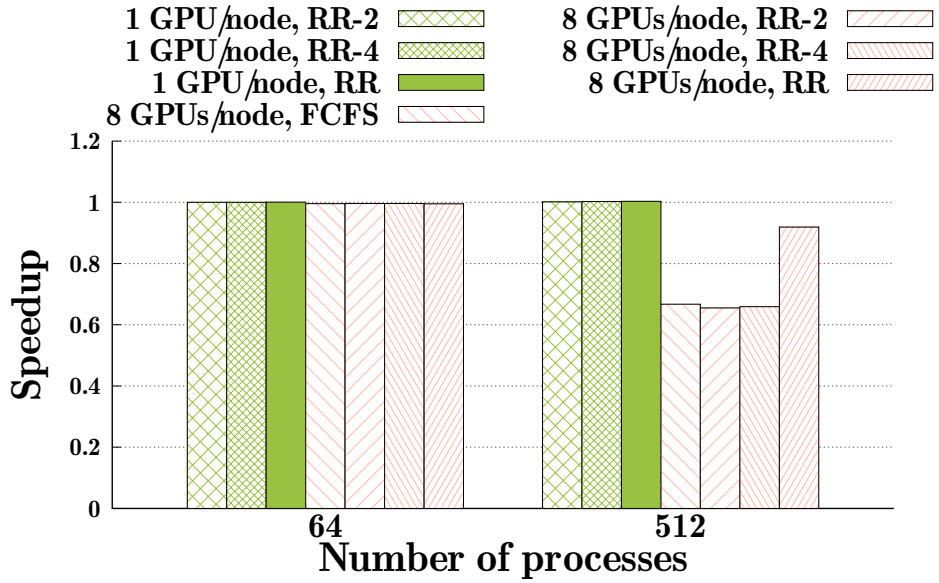

Figure 4.10: Results for Laghos (64 processes and 512 processes on 1D dragonfly)

256 processes and 128 processes), Laghos is affected by communication only in the 8 GPUs per node setting.

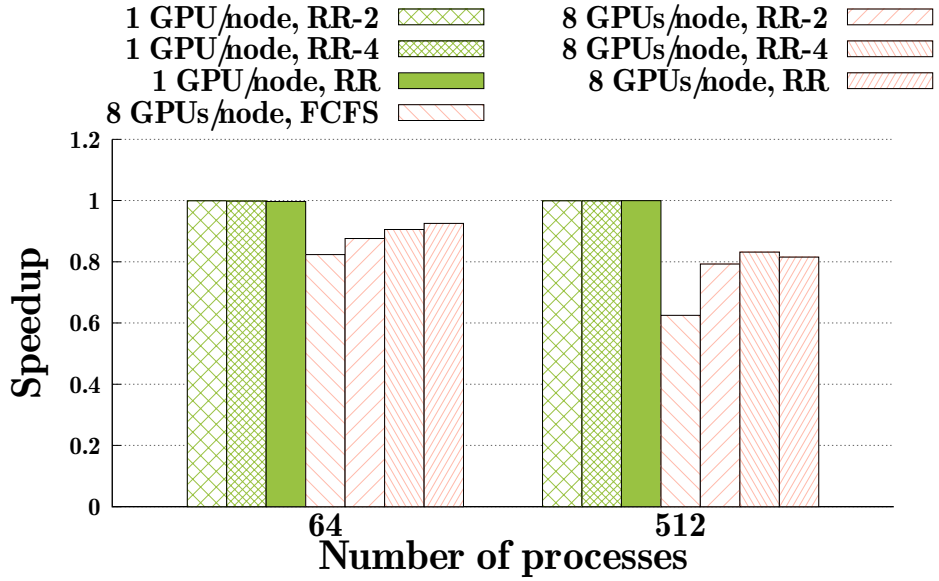

Figure 4.11: Results for SW4lite (64 processes and 512 processes on 1D dragonfly)

Figures 4.11 show the results for SW4lite. Message scheduling has no impact on the 1 GPU per node cases, but affects the performance significantly for 8 GPUs per node cases for both applications

and the two job sizes. The impact, however, depends on both the application and job sizes. Similar results are seen for AMG application.

***Overall Observation**: For most applications, some degree of round robin in NIC scheduling is effective. However the exact degree is application dependent – no single scheduling scheme can achieve the best performance across applications.*

Message scheduling can impact performance only when there are many concurrent communicating pairs. For the 1 GPU per node cases, it thus only affects the communication heavy applications such as Stencil4d, and has virtually no impact on the other applications in the study. As the number of GPUs per node increases, so does the number of communication sources and the number of concurrent communications. Thus, with 8 GPUs per node, message scheduling makes a difference in all applications, except Kripke. The magnitude of the impact, however, depends on the application as well as the job size: Round-robin (RR) is the most effective scheduling in many cases. However, each of the scheduling schemes achieves the best performance in some cases. For example, FCFS is the best for AMG with 64 processes and 8 GPUs per node; RR-4 is slightly better than other scheduling policies for SW4lite with 512 processes and 8 GPUs per node. The effectiveness of message scheduling depends on both application and the network parameters, and needs to be further studied by examining more applications as well as system configurations.

4.5 Summary

This chapter explored the intricate dynamics of optimizing hardware parameters for future GPU-based High Performance Computing (HPC) platforms. The study identified that the integration of multiple GPUs per compute node, a trend driven by the need for increased computational capacity, necessitates a reevaluation of traditional hardware configurations. The core of this investigation is the development of a comprehensive simulation study using the TraceR-CODES tool, focusing on three pivotal hardware parameters: the number of GPUs per node, network link bandwidth, and network interface controller (NIC) scheduling policies. This study is contextualized within the frameworks of two prevalent network topologies: fat-tree and dragonfly. The findings from this research challenge the conventional wisdom on HPC platform optimization. It is revealed that the interplay between hardware parameters and application performance is nuanced, requiring a tailored approach to system design. Particularly, the study unveils that the optimal configuration of GPUs per node significantly hinges on the specific demands of the applications running on

the platform. For communication-intensive applications, enhancing network bandwidth is critical for sustaining performance as the number of GPUs per node increases. Conversely, computation-heavy applications exhibit a different sensitivity pattern to network bandwidth variations but are influenced by the strategies employed for NIC scheduling. Moreover, the work introduces a novel perspective on the role of hardware parameters in shaping HPC system performance. The simulation results indicate that a holistic approach, which meticulously balances computational capacity with communication efficiency through strategic hardware parameter configuration, is imperative for the design of future GPU-based HPC platforms. By dissecting the complex relationship between hardware parameters and system performance, it lays the groundwork for designing more powerful, efficient, and capable HPC platforms, thereby addressing the evolving demands of high-performance computing applications.

CHAPTER 5

CONCLUSION

In my dissertation research, I focus on two critical areas in HPC: integrating SDN techniques to enhance HPC application performance and optimizing hardware parameters for next-generation GPU-based platforms.

In the first part of this study, I articulate an SDN-based HPC system that integrates flow identification, phase detection, and SDN routing to mitigate contention in contemporary fat-tree interconnects. I have devised two complementary flow-classification techniques: (i) a lightweight, deep-learning-based detector that recognizes elephant flows within a single SDN polling interval, and (ii) an application-level API through which HPC codes can label their own flows—an approach that eliminates the multi-second delays typical of data-centre polling. I have paired these classifiers with both explicit user phase markers and a dynamic detector so that network resources are released the instant an application transitions between communication and computation phases. With accurate, phase-aware demand profiles in hand, I have implemented three routing schemes. SDN-greedy spreads traffic by minimising the worst-link load; SDN-optimal leverages a graph-theoretic edge-colouring to deliver contention-free schedules on full-bisection fat-trees; and I have modified SDN-optimal to operate on 3-to-1 tapered fat-trees and introduce SDN-adaptive, a multi-path solution for scenarios in which traditional adaptive routing underperforms. Simulation results demonstrate that these routing strategies reduce congestion and improve communication performance in an application-dependent manner, highlighting the importance of SDN-based techniques such as programmable routing in next-generation HPC systems.

In the second part, I explore optimizing hardware parameters for GPU-based HPC platforms. With the current trend of HPC systems moving toward higher computational capacity GPU nodes, it is essential to evaluate the impact of key hardware design parameters—such as the number of GPUs per node, network link bandwidth, and network interface controller (NIC) scheduling policies—within fat-tree and dragonfly topologies. Using the TraceR-CODES simulation tool, I have analyzed the effects of these parameters on the computation and communication capacities for various HPC applications. The results indicate that as more GPUs are integrated per node, the sensitivity of applications to communication performance increases, necessitating higher network

bandwidth and effective scheduling methods to maintain optimal system performance. The exact impact of these hardware parameters is application-dependent, highlighting the need for tailored investigations to determine cost-effective configurations.

In summary, my dissertation research contributes to optimizing HPC platforms by addressing both networking and hardware challenges. By utilizing SDN and enhancing GPU integration for better network management, I provide practical solutions for developing next-generation HPC systems that achieve optimal performance and efficiency.

REFERENCES

[1] Bilge Acun et al. "Preliminary Evaluation of a Parallel Trace Replay Tool for HPC Network Simulations." In: *Euro-Par 2015: Parallel Processing Workshops*. Ed. by Sascha Hunold et al. Cham: Springer International Publishing, 2015, pp. 417–429. ISBN: 978-3-319-27308-2.

[2] Yehuda Afek et al. "Sampling and large flow detection in SDN." In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 2015, pp. 345–346.

[3] Shiyam Alalmaei et al. "SDN heading north: Towards a declarative intent-based northbound interface." In: *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE. 2020, pp. 1–5.

[4] Zaid ALzaid, Saptarshi Bhowmik, and Xin Yuan. "Multi-path routing in the jellyfish network." In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2021, pp. 832–841.

[5] Zaid Salamah A Alzaid et al. "Global Link Arrangement for Practical Dragonfly." In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, 2020. ISBN: 9781450379830. DOI: 10.1145/3392717.3392756. URL: https://doi.org/10.1145/3392717.3392756.

[6] Omer Arap et al. "Software defined multicasting for mpi collective operation offloading with the netfpga." In: *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings 20*. Springer. 2014, pp. 632–643.

[7] Billy Joe Archer and Benny Manuel Vigil. *The trinity system*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2015.

[8] SDN architecture. *Tr, ONF*. https://opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf. 2016.

[9] Kamal Benzekki, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. "Software-defined networking (SDN): a survey." In: *Security and communication networks* 9.18 (2016), pp. 5803–5833.

[10] A Bhatele. *Evaluating trade-offs in potential exascale interconnect topologies*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[11] Abhinav Bhatele et al. "Analyzing cost-performance tradeoffs of HPC network designs under different constraints using simulations." In: *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 2019, pp. 1–12.

[12] Peter J Cameron. "Hall's marriage theorem." In: *arXiv preprint arXiv:2503.23159* (2025).

[13] Mosharaf Chowdhury and Ion Stoica. "Coflow: A Networking Abstraction for Cluster Applications." In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. HotNets-XI. Redmond, Washington: Association for Computing Machinery, 2012, p. 3136. ISBN: 9781450317764. DOI: 10.1145/2390231.2390237. URL: https://doi.org/10.1145/2390231.2390237.

[14] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[15] Daniele De Sensi et al. "An in-depth analysis of the slingshot interconnect." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–14.

[16] Greg Faanes et al. "Cray cascade: a scalable HPC system based on a Dragonfly network." In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–9.

[17] Peyman Faizian et al. "A comparative study of SDN and adaptive routing on dragonfly networks." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–11.

[18] Ahmad Faraj and Xin Yuan. "Communication characteristics in the NAS parallel benchmarks." In: *IASTED PDCS*. 2002, pp. 724–729.

[19] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A scalable, commodity data center network architecture." In: *ACM SIGCOMM computer communication review* 38.4 (2008), pp. 63–74.

[20] Mohammad Al-Fares et al. "Hedera: dynamic flow scheduling for data center networks." In: *Nsdi*. Vol. 10. 8. San Jose, USA. 2010, pp. 89–92.

[21] Open Networking Foundation. *OpenFlow Switch Specification*. https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf. 2015.

[22] Richard M Fujimoto. "Parallel discrete event simulation." In: *Communications of the ACM* 33.10 (1990), pp. 30–53.

[23] Crispn Gomez et al. "Deterministic versus adaptive routing in fat-trees." In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, pp. 1–8.

[24] Jing Gong et al. "Nekbone performance on GPUs with OpenACC and CUDA Fortran implementations." In: *The Journal of Supercomputing* 72 (2016), pp. 4160–4180.

[25] Steven Gottlieb and Sonali Tamhankar. "Benchmarking MILC code with OpenMP and MPI." In: *Nuclear Physics B-Proceedings Supplements* 94.1-3 (2001), pp. 841–845.

[26] Philip Hall. "On representatives of subsets." In: *Classic Papers in Combinatorics* (1987), pp. 58–62.

[27] Emily Hastings et al. “Comparing global link arrangements for dragonfly networks.” In: *2015 IEEE International Conference on Cluster Computing*. IEEE. 2015, pp. 361–370.

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

[29] Chi-Yao Hong et al. “Achieving high utilization with software-driven WAN.” In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. 2013, pp. 15–26.

[30] Nikhil Jain et al. “Evaluating HPC networks via simulation of parallel workloads.” In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, pp. 154–165.

[31] Nikhil Jain et al. “Predicting the Performance Impact of Different Fat-Tree Configurations.” In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126967. URL: https://doi.org/10.1145/3126908.3126967.

[32] Nikhil Jain et al. “Predicting the performance impact of different fat-tree configurations.” In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–13.

[33] Fulya Kaplan et al. “Unveiling the interplay between global link arrangements and network management algorithms on dragonfly networks.” In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2017, pp. 325–334.

[34] John Kim et al. “Technology-Driven, Highly-Scalable Dragonfly Topology.” In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. USA: IEEE Computer Society, 2008, pp. 77–88. ISBN: 9780769531748. DOI: 10.1109/ISCA.2008.19. URL: https://doi.org/10.1109/ISCA.2008.19.

[35] John Kim et al. “Technology-driven, highly-scalable dragonfly topology.” In: *ACM SIGARCH Computer Architecture News* 36.3 (2008), pp. 77–88.

[36] Andreas Knüpfer et al. “Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir.” In: *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[37] Dénes König. “Über graphen und ihre anwendung auf determinantentheorie und mengenlehre.” In: *Mathematische Annalen* 77.4 (1916), pp. 453–465.

[38] Diego Kreutz et al. “Software-defined networking: A comprehensive survey.” In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[39] Charles E Leiserson. "Fat-trees: universal networks for hardware-efficient supercomputing." In: *IEEE transactions on Computers* 100.10 (1985), pp. 892–901.

[40] Shang Li et al. "Low latency, high bisection-bandwidth networks for exascale memory systems." In: *Proceedings of the Second International Symposium on Memory Systems*. 2016, pp. 62–73.

[41] LLNL. *Kripke*. `https://computing.llnl.gov/projects/co-design/kripke`. 2020.

[42] LLNL. *Quartz*. `https://hpc.llnl.gov/hardware/platforms/Quartz`. 2020.

[43] LLNL. *Sequoia*. `https://asc.llnl.gov/computers/historic-decommissioned-machines/sequoia-and-vulcan`. 2020.

[44] LLNL. *Laghos*. `https://computing.llnl.gov/projects/co-design/laghos`. 2020.

[45] LLNL. *Vulcan*. `https://asc.llnl.gov/computers/historicdecommissioned-machines/vulcan`. 2020.

[46] Santosh Mahapatra, Xin Yuan, and Wickus Nienaber. "Limited multi-path routing on extended generalized fat-trees." In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE. 2012, pp. 938–945.

[47] Misbah Mubarak et al. "Enabling parallel simulation of large-scale HPC network systems." In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2016), pp. 87–100.

[48] Baatarsuren Munkhdorj et al. "Design and implementation of control sequence generator for sdn-enhanced mpi." In: *Proceedings of the Fifth International Workshop on Network-Aware Data Management*. 2015, pp. 1–9.

[49] Wickus Nienaber. "Effective Routing on Fat-Tree Topologies." In: (2014).

[50] Titan at OLCF web page. *Titan*. `https://www.olcf.ornl.gov/titan/`. 2022.

[51] ORNL. *Summit*. `https://www.olcf.ornl.gov/summit/`. 2020.

[52] John D Owens et al. "GPU computing." In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[53] MC Paull. "Reswitching of connection networks." In: *Bell System Technical Journal* 41.3 (1962), pp. 833–855.

[54] Delivering Better Price and Performance than Ethernet. "InfiniBand Clustering." In: ().

[55] Bogdan Prisacari et al. "Bandwidth-optimal all-to-all exchanges in fat tree networks." In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 2013, pp. 139–148.

[56] ECP Proxy. *AMG*. https://proxyapps.exascaleproject.org/app/amg/. 2020.

[57] Md Shafayat Rahman et al. "Topology-custom UGAL routing on dragonfly." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–15.

[58] German Rodriguez et al. "Exploring pattern-aware routing in generalized fat tree networks." In: *Proceedings of the 23rd international conference on Supercomputing*. 2009, pp. 276–285.

[59] German Rodriguez et al. "Oblivious routing schemes in extended generalized fat tree networks." In: *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE. 2009, pp. 1–8.

[60] Christian Scheideler. *Universal routing strategies for interconnection networks*. Vol. 1390. Springer, 2006.

[61] Bjorn Sjogreen. *SW4 final report for iCOE*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[62] Junho Suh et al. "Opensample: A low-latency, sampling-based measurement platform for commodity sdn." In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE. 2014, pp. 228–237.

[63] Philip Taffet et al. "Testing the Limits of Tapered Fat Tree Networks." In: *Proceedings of the IEEE/ACM Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at SC 2019*. 2019, pp. 47–53. DOI: 10.1109/PMBS49563.2019.00011.

[64] Keichi Takahashi et al. "Concept and design of sdn-enhanced mpi framework." In: *2015 Fourth European Workshop on Software Defined Networks*. IEEE. 2015, pp. 109–110.

[65] Keichi Takahashi et al. "Performance evaluation of SDN-enhanced MPI allreduce on a cluster system with fat-tree interconnect." In: *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2014, pp. 784–792.

[66] Aidan P Thompson et al. "LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales." In: *Computer Physics Communications* 271 (2022), p. 108171.

[67] Yang Xu et al. "Identifying SDN state inconsistency in OpenStack." In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015, pp. 1–7.

[68] Ze Yang and Kwan L Yeung. "Flow monitoring scheme design in SDN." In: *Computer Networks* 167 (2020), p. 107007.

[69] Eitan Zahavi et al. “Optimized InfiniBandTM fat-tree routing for shift all-to-all communication patterns.” In: *Concurrency and Computation: Practice and Experience* 22.2 (2010), pp. 217–231.

BIOGRAPHICAL SKETCH

The author was born in India and earned a Master of Science degree in Computer Science from Jadavpur University in Kolkata. After working in the industry and completing a research internship at VMware, the author moved to the United States to pursue a Ph.D. in Computer Science at Florida State University. His research focuses on high-performance computing networks, software-defined networking, and performance modeling of parallel applications. During his doctoral studies, he has interned at Lawrence Livermore National Laboratory and contributed to the development of simulation frameworks used for evaluating HPC systems.