FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

DESIGN AND EVAUATION 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 3, 2025.
The members of the supervisory committee were:

Xin Yuan
Professor Directing Thesis

Metcalfe
University Representative

Weikuan Yu
Committee Member

Gary Tyson
Committee Member

Abhinav Bhatele
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.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The demand for High-Performance Computing (HPC) applications has surged due to the increasing complexity of scientific computations. The introduction of new GPU based compute nodes and SDN technology 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 more powerful GPU-based compute nodes, communication fails to scale proportionally. New technologies like Software-Defined Networking (SDN) attempt to bridge this gap by providing improved resource management during communication. Still these challenges prevails, which in turn necessitate thorough investigation, thus prompting the need for research on how applications perform on machines equipped with new technologies. This prospectus proposes leveraging SDN in conjunction with GPU acceleration to enhance communication efficiency of applications in HPC environments and also find the optimal system configuration for running different applications in these environments. By alleviating communication bottlenecks, this research aims to facilitate seamless execution of HPC workloads, thereby enabling groundbreaking discoveries in scientific computing.

CHAPTER 1

INTRODUCTION

Large-scale systems aiming to achieve over 1 Exaflop/s of sustained performance are currently under construction. 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 [32], 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 [39], 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 [40]. 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 applications 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.

Software Defined Networking (SDN) [28] 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 map-reduce applications [20], 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.

Our research primarily focuses on the following:

- Understanding the performance implication of technology shifts. 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 change in these parameters.
- Developing and evaluating SDN techniques for HPC systems/applications. 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.

The structure of this prospectus aligns with the outlined research objectives. Chapter 2 will provide a background on interconnection technologies, SDN fundamentals and related works. In chapter 3, I will examine the influence of network parameters on modern HPC systems. Chapter 4 will delve into the implementation of SDN enhancements within HPC environments and evaluate their impact on application performance. Finally, chapter 5 will offer concluding remarks.

CHAPTER 2

BACKGROUND AND RELATED WORKS

In the domain of High-Performancell 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). By covering these fundamental concepts, readers will gain 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 usually shown as a graph, where each point (vertex) represents a piece of hardware like a server or a switch, and each line (edge) represents a connection between them. I usually refer to servers as processing elements, and switches or routers as forwarding elements. When two points are directly connected by a line, I say they are neighbors. The number of connections a point has is called its nodal degree. The distance between two points is how many connections (or hops) it takes to get from one to the other. The diameter of a network is the longest distance between any two points. 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. The bisection bandwidth is the lowest possible bandwidth among all possible splits. If the bandwidth is low, it can slow down traffic. For networks used in HPC and data centers, I want a low diameter and high bisection bandwidth to perform well at scale. I also aim for a low nodal degree to keep costs down and avoid complex designs. 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 [29]. 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. Hierarchy and Switch Types: 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.
- **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.

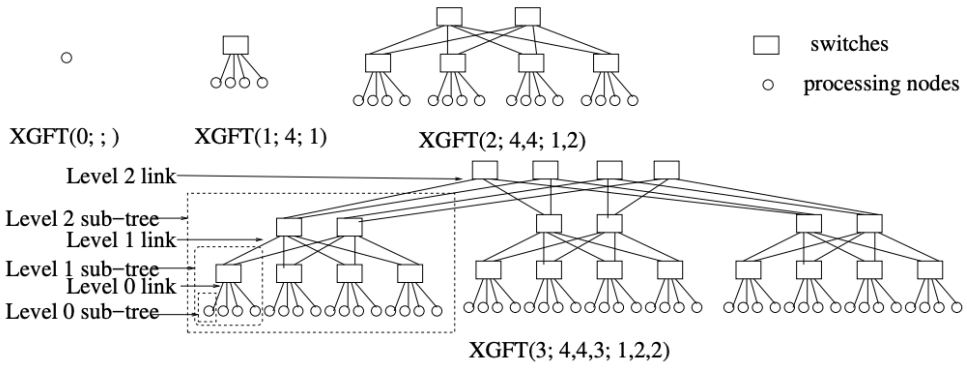

Figure 2.1: A fat-tree represented in XGFT.

Fat-tree topology ensures that the bandwidth within a level remains consistent, if not increasing, as one moves toward higher levels [29, 23]. A full bisection bandwidth fat-tree is a type of network

configuration designed to ensure that every node on one half of the network can communicate with every other node in the other half of the network without creating bottlenecks. In this setup, the bandwidth between any two points in the network is maximized, which helps to prevent congestion and ensures smooth communication. To ensure full bisection bandwidth, more network resources are allocated, which increases the cost. One method used to reduce the cost of building a fat-tree network is tapering. Tapering involves connecting more devices to each switch at the lower levels of the network, known as leaf switches. 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 fat-tree configuration.

Full bisection bandwidth fat-tree networks can be described using two key parameters: 'm' and 'n'. The parameter 'm' signifies the degree of all internal nodes, which must be divisible by 2. Meanwhile, 'n' denotes the number of levels of internal nodes, resulting in a fat-tree with n + 1 levels in total [29]. To economize network costs, tapering strategies can be implemented, allowing for more nodes to be connected per leaf switch. For comprehensive representation and analysis of any fat-tree topologies, Ohring introduced extended generalized fat-tree (XGFT) representations [37]. These representations provide a structured method for describing fat-tree configurations, aiding in both design and analysis processes. The XGFT notation, capable of representing various fat-tree variations, specifies a fat-tree of height 'h', comprising h + 1 levels of nodes. Each level is labeled from 0 to h, starting with processing nodes at level 0. Moreover, each node at level 'i' has 'wi' parents, while each node at level 'i' has 'mi-1' children. This notation is exemplified by the recursive construction of XGFT(3; 4, 4, 3; 1, 2, 2), where 'h' equals 3, 'm0' equals 4, 'm1' equals 4, 'm2' equals 3, 'w0' equals 1, 'w1' equals 2, and 'w2' equals 2.

To ensure full bisection bandwidth, more network resources are allocated, which increases the cost. One method used to reduce the cost of building a fat-tree network is tapering. Tapering involves connecting more devices to each switch at the lower levels of the network, known as leaf switches. 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 fat-tree The tapering ratio determines the extent of uplink reduction, balancing network performance and hardware efficiency. In a 3-to-1 tapering configuration, for every one uplink from a leaf switch, three downlinks connect downward. Tapered fat-trees are commonly used in data centers and HPC clusters where traffic patterns are predictable. This allows network designers to allocate resources strategically, optimizing both costs and power consumption.

The figure below (Figure 2.2) illustrates a tapered fat-tree pod for a 3-level fat-tree with 1536 nodes and switches with 32 ports each.

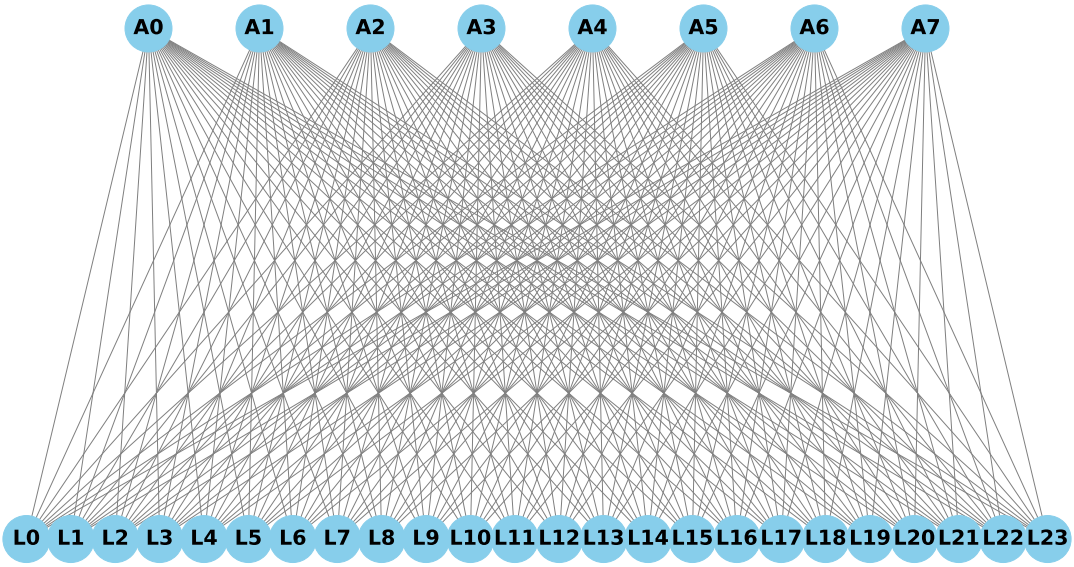

Figure 2.2: 3-to-1 Tapered Fat-tree pod

2.1.2 Dragonfly

The dragonfly topology stands out as a cost-effective solution for building expansive interconnection networks [26]. This design is characterized by its two-layer structure, exemplified in Figure 2.3. Initially proposed by Kim et al. [26], 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. In a practical scenario, such as the one illustrated in Figure 2.3, each group typically encompasses 4 switches, culminating in a total of 9 groups within the network. There are variations of dragonfly topology, including canonical dragonfly, hamming dragonfly, and dragonfly plus, which utilize various intra-group connectivity patterns [19]. However, all implementations of dragonfly topology feature all-to-all connectivity between groups. At the inter-group level, dragonfly networks consistently adopt a fully connected topology. A crucial aspect of the dragonfly topology revolves around three key parameters: the number of compute nodes in each switch (p), the intra-group links per switch (a), and the inter-group links per switch (h) [26].

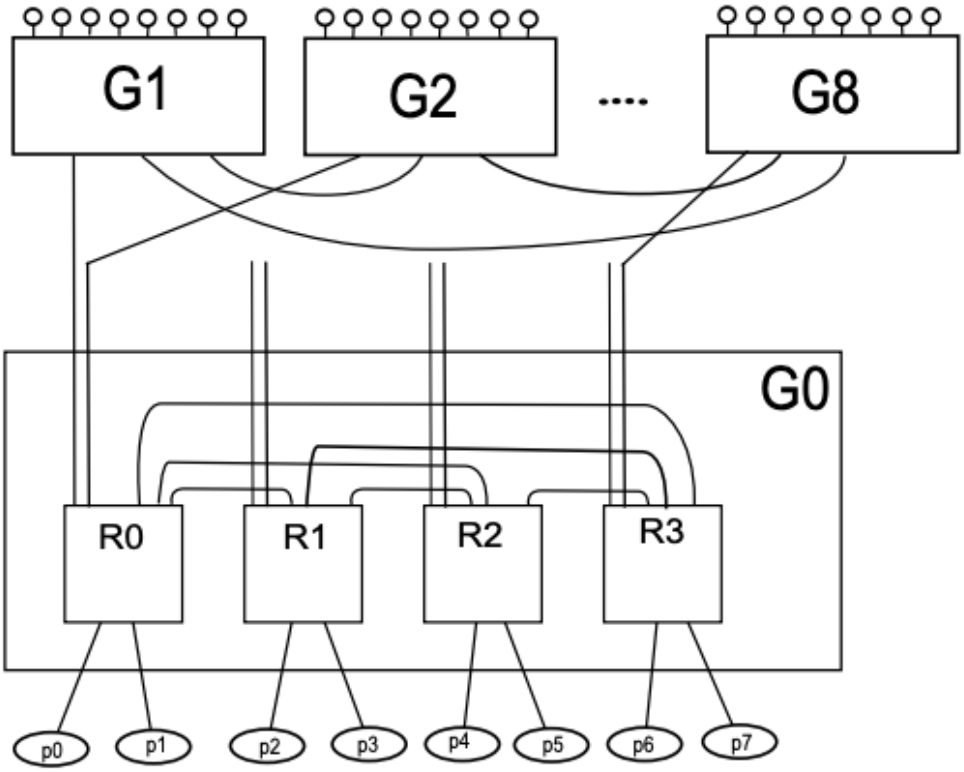

Figure 2.3: Dragonfly architecture with 9 groups and 4 routers per group

The dragonfly network comprises a total of g groups, g × a routers, and g × a × 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 × h global links, with h representing the inter-group links per switch. Notably, the maximum size dragonfly configuration is characterized by g = a × h + 1 groups. A balanced dragonfly configuration typically necessitates a = 2p = 2h, ensuring efficient load distribution across the network. Several prominent supercomputer architectures, such as the Cray Cascade architecture and the Cray Slingshot network, have embraced variations of the dragonfly topology [13, 12]. These implementations have found their way into current supercomputers like Titan [38] 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

Efficient routing plays a critical role in optimizing network flow by determining the most effective paths for data transmission. Routing in general determines how data packets, which are units of data transmitted over a network, travel from a starting point to an endpoint 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 server, while the destination switch connects to the destination server. 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. Different routing schemes cater to various interconnect designs, with four common approaches being deterministic routing, adaptive routing, source routing, and hop-by-hop routing. Deterministic routing precomputes paths for each source-destination pair, with packets consistently sent along these predetermined paths. Adaptive routing dynamically selects paths based on the current network state, allowing for real-time adjustments to optimize performance. Source routing involves the source node determining the complete path for each packet and embedding this routing information within the packet header, while hop-by-hop routing allows each intermediate node along the path to independently make 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 oblivious routing schemes in fat-tree networks are Source-mod-k (S-mod-k) and Destination-mod-k (D-mod-k). Both S-mod-k and D-mod-k are considered equivalent in terms of conceptual design and performance. However, the D-mod-k algorithm may exhibit poor performance for both average and worst-case permutation traffic patterns. In contrast, adaptive routing algorithms dynamically select paths based on the current network state, such as link congestion or available bandwidth,

to optimize performance in real-time. In adaptive routing within fat-tree networks, the routing strategy dynamically selects 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. 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. First, I have the Minimal Routing (MIN) scheme [26], which directs data packets exclusively along the shortest routes between source and destination nodes. Minimal paths typically involve traversing one global link at most. MIN routing aims to minimize resource usage and is effective for traffic patterns such as random uniform traffic. Suppose I have a dragonfly network topology where a source node 'S' in group A needs to communicate with a destination node 'D' in group B. In MIN routing, the data packet would follow the shortest path, traversing one global link at most. For example, the packet might first travel from 'S' to a switch in group B via a global link, then to 'D' within group B. Second, I have Valiant Load-Balanced Routing (VLB) [26, 24], which spreads non-uniform traffic evenly across available links to mitigate congestion. It involves routing from the source to an intermediate switch, then to the destination. VLB paths are non-minimal and aim to balance traffic distribution across the network. In VLB routing, the data packet would be routed from 'S' to an intermediate switch 'I', which is not present in either source group or destination group, then to 'D'. For instance, the packet might travel from 'S' to 'I' via a global link, then from 'I' to 'D' through a global link. One of the drawbacks of VLB routing is that it increases the hop count for the transmission of data. Finally, I have Universal Globally

Adaptive Load-Balanced Routing (UGAL) [26, 24], which dynamically selects between MIN and VLB paths based on network conditions. UGAL utilizes the buffer occupancy in the source router to estimate network congestion in real-time. This means that UGAL monitors the amount of data packets queued up or waiting to be transmitted at the source router. By assessing the level of buffer occupancy, UGAL can infer the current state of congestion within the network. If the buffer occupancy is high, indicating congestion, UGAL may opt for routing strategies that help alleviate congestion, such as selecting Valiant Load-Balanced (VLB) paths to distribute traffic more evenly across available links. It chooses a path with the smallest packet delay from a small number of candidate MIN and VLB paths. For example, if the network senses congestion on minimal paths, it might choose a VLB path for certain packets to balance the traffic load. Conversely, if the network conditions allow, it might opt for a minimal path to minimize latency.

2.3 HPC Applications

In the realm of High-Performance Computing (HPC), applications span a wide spectrum, including real-world problem-solving (a.k.a. real applications), proxy modeling (a.k.a. proxy applications), and synthetic traffic. Whether simulating climate patterns, optimizing financial portfolios, or deciphering genomic data, all 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 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, 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.
- **Stencil4d**: MPI benchmark with 8-point near-neighbor communication in a 4D virtual process grid.
- **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 [30].
- **Laghos**: Proxy application that solves time-dependent Euler equations with MPI-based domain decomposition [33].
- **AMG**: Parallel algebraic multigrid solver [41].
- **SW4lite**: Proxy application for SW4 [43], 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 [47].
- **Nekbone**: Solves 3D Poisson problem in rectangular geometry. The key MPI operations are matrix-matrix multiplication, inner products, nearest neighbor communicaton, MPI_Allreduce [17].
- **MILC**: Performs four dimensional SU(3) lattice gauge theory, mainly through near-neighbour communication and MPI_Allreduce [18].

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, or static, 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 2.4 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:** Irregular communication in HPC applications includes both dynamic and dynamically analyzable communication patterns. 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 occured. Figure 2.5 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 2.4: Stencil4d code snippet

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

Figure 2.5: 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 [28]. Each layer serves a critical role in facilitating the efficient operation and management of the network infrastructure.

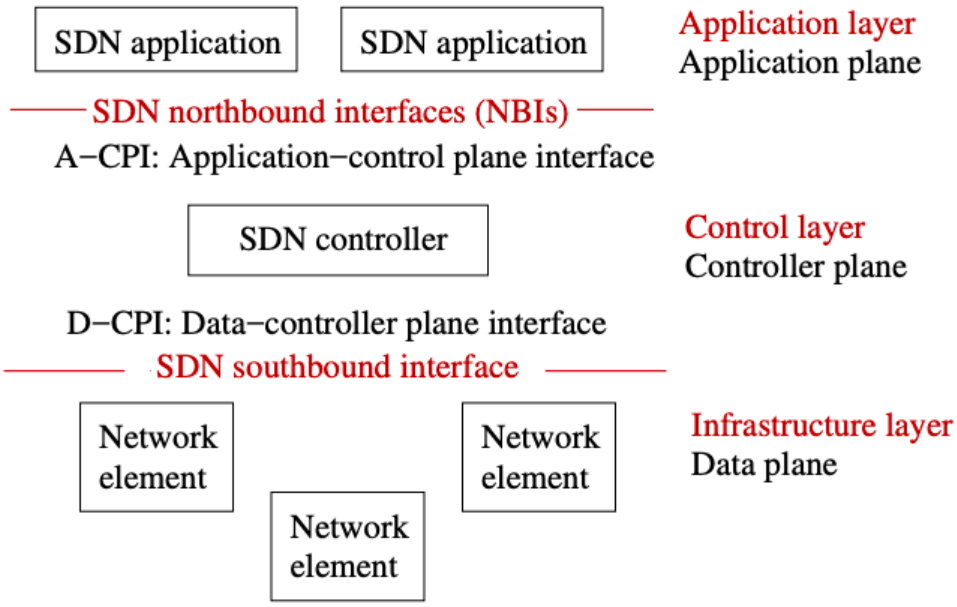

Figure 2.6: SDN abstraction

- **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.

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, 28].
- **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, 28].
- **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, 28].

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, 14]. 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.

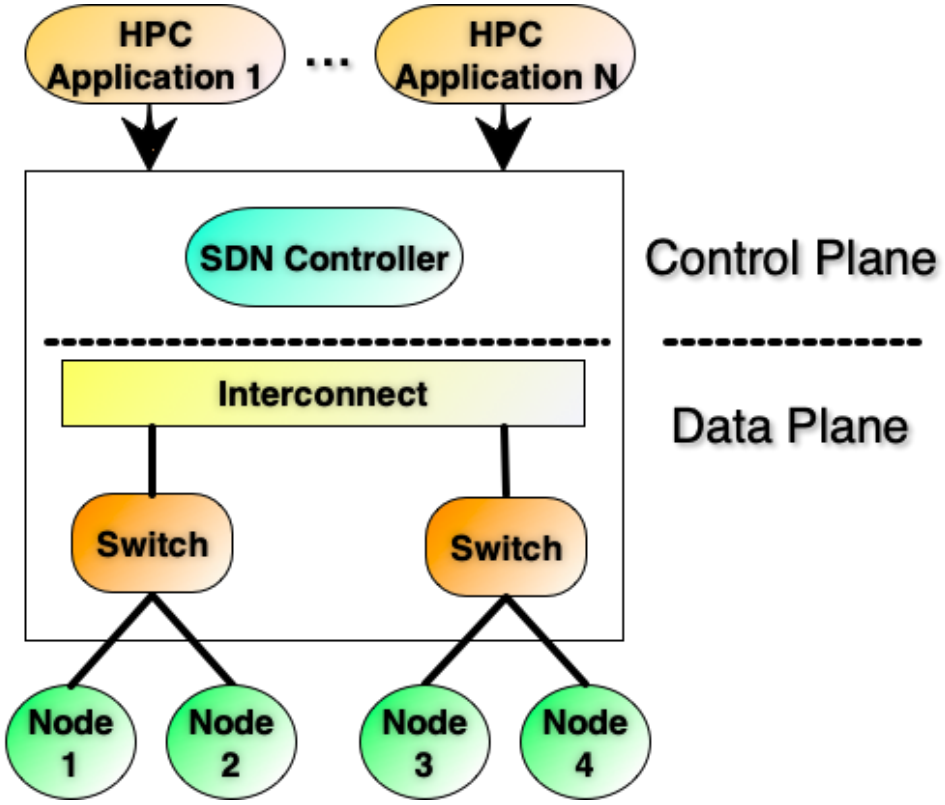

Figure 2.7: High-level overview of the SDN-based HPC System

The structure of an SDN based HPC System (SHS) is depicted in Figure 2.7. 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.

2.5 Related Works

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 [23]. This work highlights the critical role of network architecture in HPC systems, especially as more GPUs are integrated per node.

In my dissertation, I build on these studies by focusing on two areas: optimizing hardware parameters for GPU-based HPC platforms and incorporating Software Defined Networking (SDN) to improve HPC application performance. In the first 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 [23] and studies on dragonfly and jellyfish topologies by Rahman et al. [42] and Zaid et al. [4], I investigate how different hardware parameters affect network performance.

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 the second section, I analyze the integration of SDN in HPC environments. 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 [14]. Since the introduction of SDN and OpenFlow [16], 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 [46]. 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 [36, 45]. In sum-

mary, my research optimizes HPC platforms by bridging the gap between computational capacity and communication efficiency and integrating advanced networking paradigms like SDN. These efforts aim to develop more powerful, efficient, and capable HPC systems for next-generation HPC workloads.

The remainder of this prospectus will first delve into the specific challenges and opportunities associated with integrating SDN into HPC environments. It will explore novel approaches to optimizing routing, with the goal of enhancing the performance and scalability of state-of-the-art HPC environments in the era of SDN.

CHAPTER 3

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, network link bandwidth, 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, network bandwidth, 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.

In summary, this work contributes to the ongoing process of optimizing HPC platforms for the era of GPU acceleration, offering insights that can inform both the design and implementation of future systems. By bridging the gap between computational capacity and communication efficiency, this research aims to pave the way for more powerful, efficient, and capable HPC platforms.

3.1 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 [22]. 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 [22, 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 [31] 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 [34], 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.

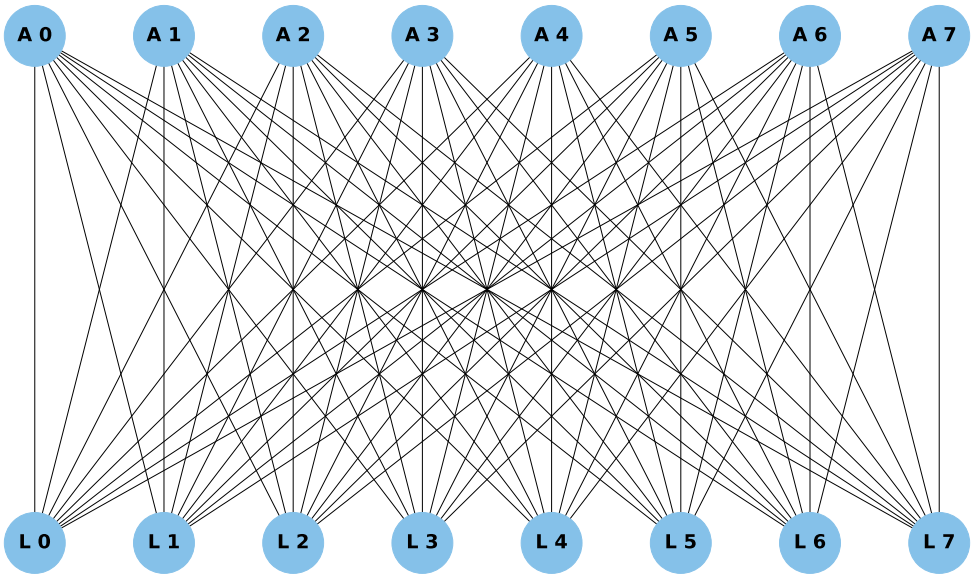

Figure 3.1: A Quartz pod with eight aggregate and eight leaf switches, and all links.

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 3.1 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 3.2 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.

3.2 Simulating Message Scheduling in the NIC

The network interface controller (NIC) in contemporary HPC systems is responsible for scheduling and packetizing messages.

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

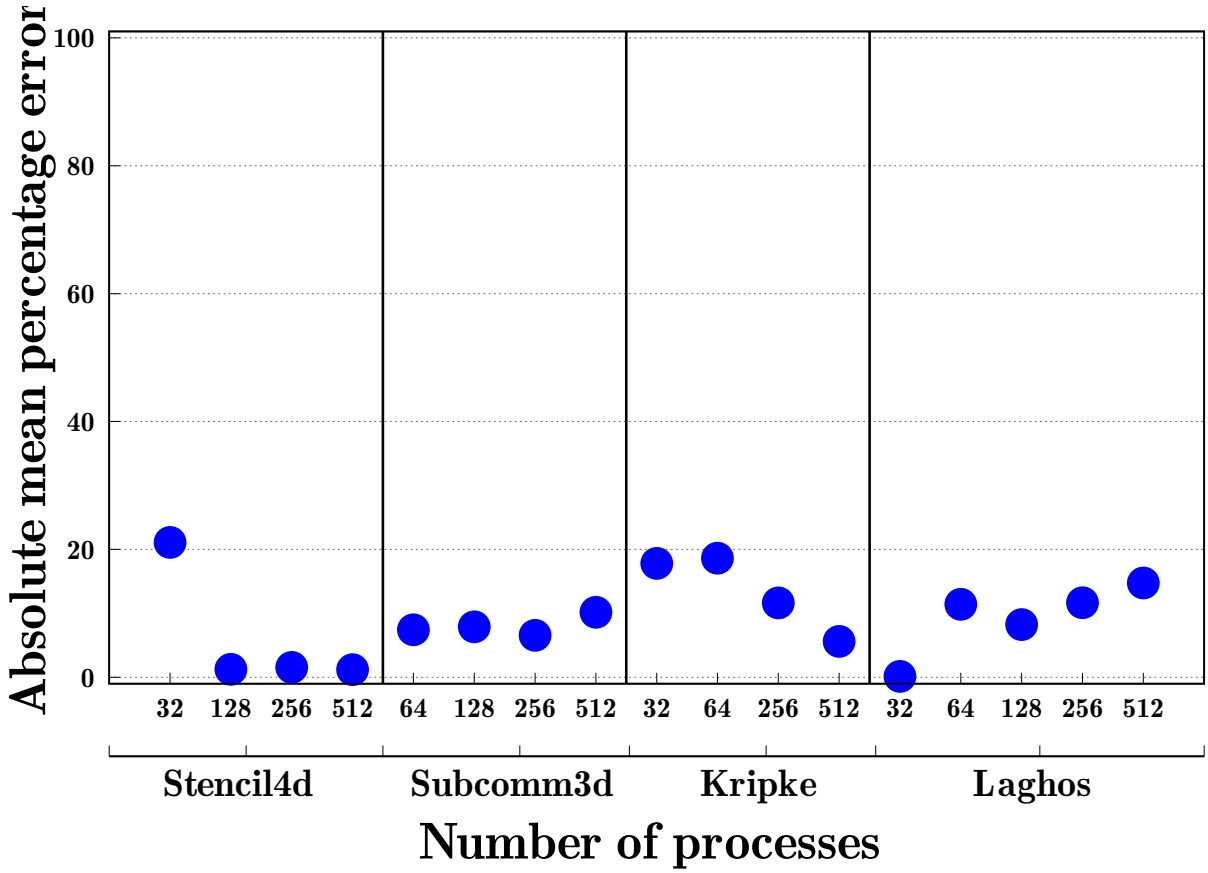

Figure 3.2: Validation of TraceR-CODES (mean percentage error in predicted runtime compared to the actual runtime).

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.

3.3 Hardware Design 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 [25] 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 [25]: 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) [25, 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

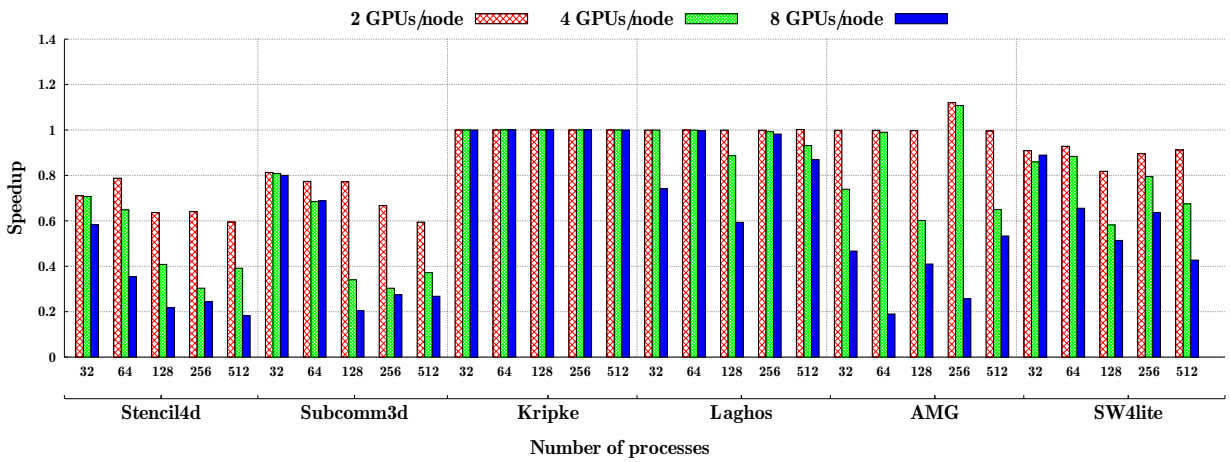

Figure 3.3: Speedup on fat-tree for various numbers of GPUs per node settings with respect to 1 GPU/node configuration.

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 3.1 summarizes the network sizes used for each GPU per node setting, with the default setting being that of 1 GPU per node.

Table 3.1: 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 paper, 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 perfor-

mance. 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.

3.4 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[30] and Laghos[33], and two applications with a balanced communication-to-computation ratio, AMG[41] and SW4lite[43] (see Figure 3.4). The traces used in the study were collected using Score-P [27] 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 3.4 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 3.4 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 [21]. 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.

3.5 Performance Study

The results of simulation studies of various different architectural parameters which are prerepresented in Section 3.3 are shown below.

3.5.1 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 3.3 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 3.3, 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 3.3, 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 3.5. 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.

3.5.2 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 3.6 shows the performance for the 4 GPUs per node configuration with varying 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 3.7.

Table 3.2 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 3.2: 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.

3.5.3 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 3.8 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 3.9 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 3.10 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 3.5, for 512 processes (and 256 processes and 128 processes), Laghos is affected by communication only in the 8 GPUs per node setting.

Figures 3.11 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. Simillar 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.

3.6 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. In the following chapter, I apply this groundwork and select specific hardware parameters which can optimally run HPC applications in a state-of-the-art HPC environment for evaluating SDN-enhanced routings.

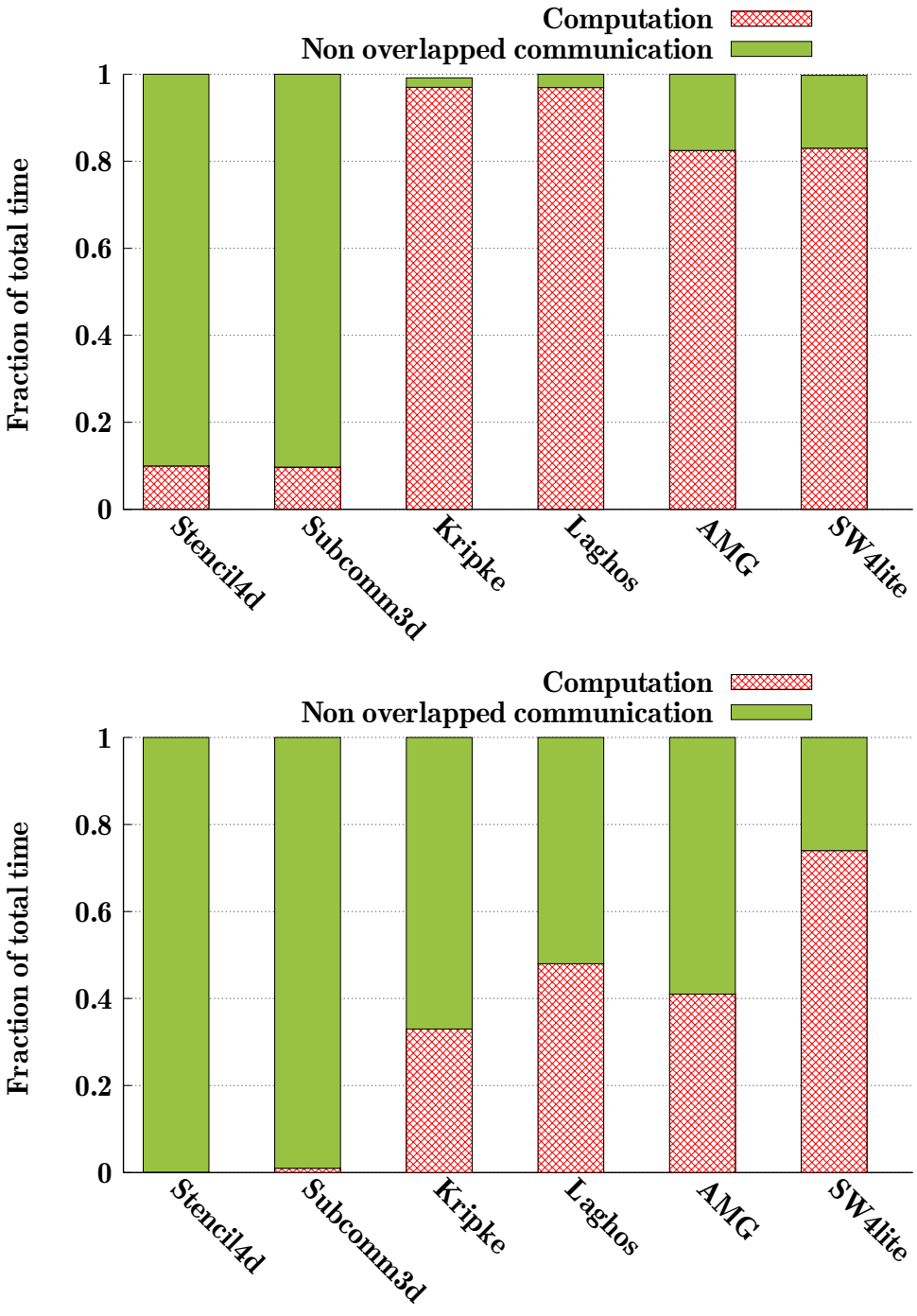

Figure 3.4: Computation and communication characteristics of all applications without scaling (left) and with scaling for GPUs (right) running on 32 processes.

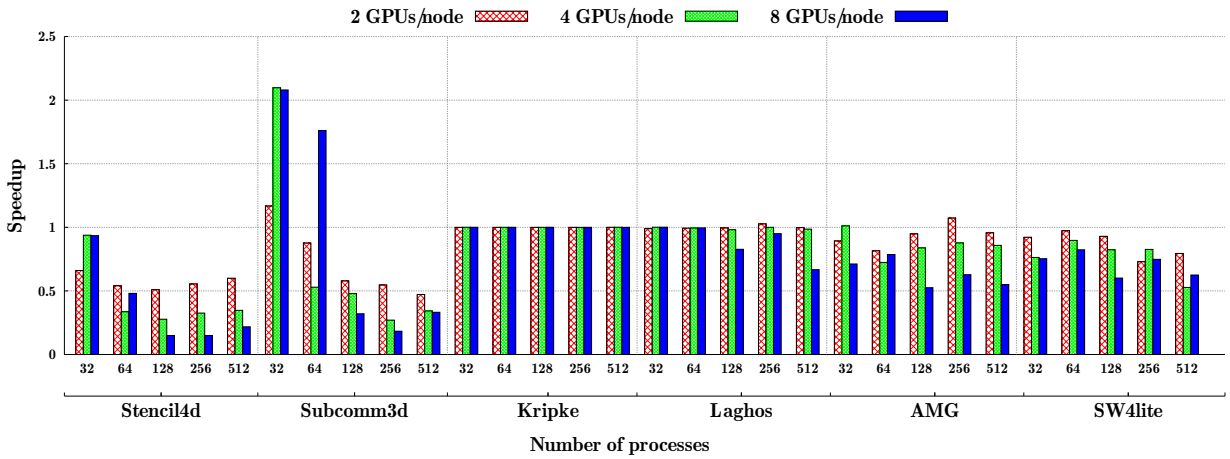

Figure 3.5: Speedup on 1D dragonfly for various numbers of GPUs per node settings with respect to 1 GPU/node configuration.

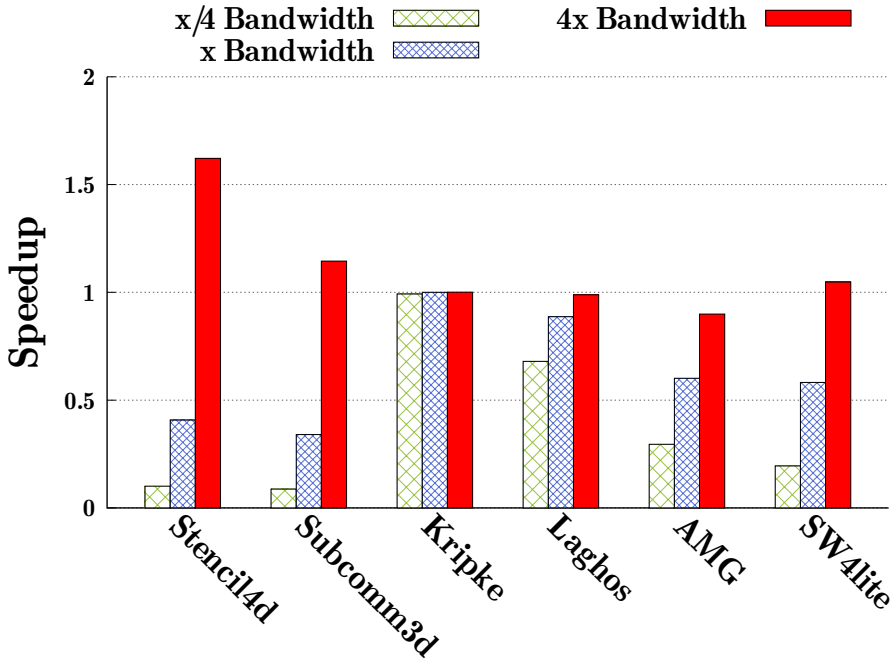

Figure 3.6: Speedup for the 4 GPUs/node configuration over 1 GPU/node in fat-tree, 1x network bandwidth configuration. Data is shown only for job sizes of 128 GPUs.

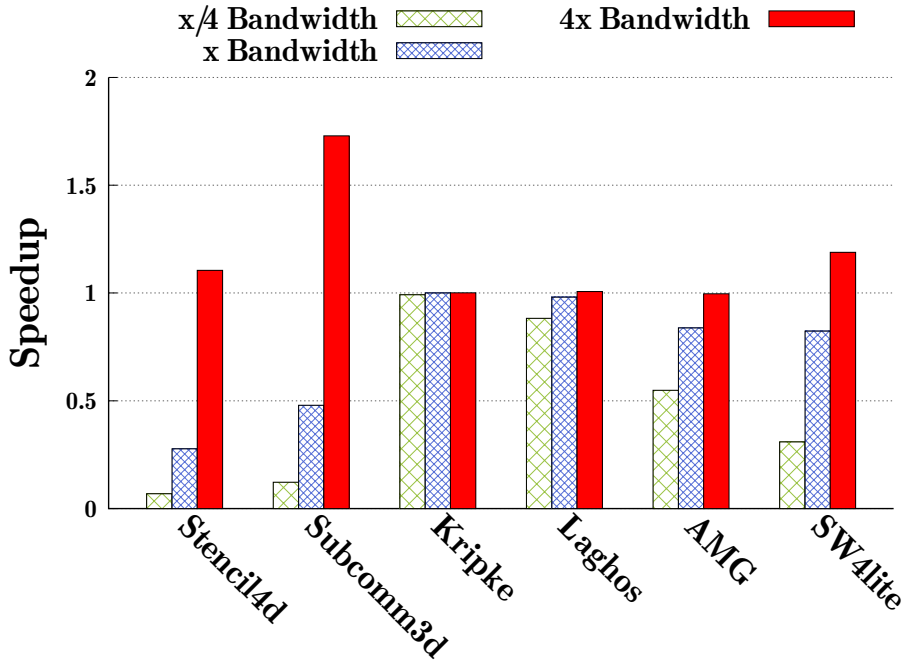

Figure 3.7: Speedup for the 4 GPUs/node configuration over 1 GPU/node in 1D dragonfly, 1x network bandwidth configuration. Data is shown only for job sizes of 128 GPUs.

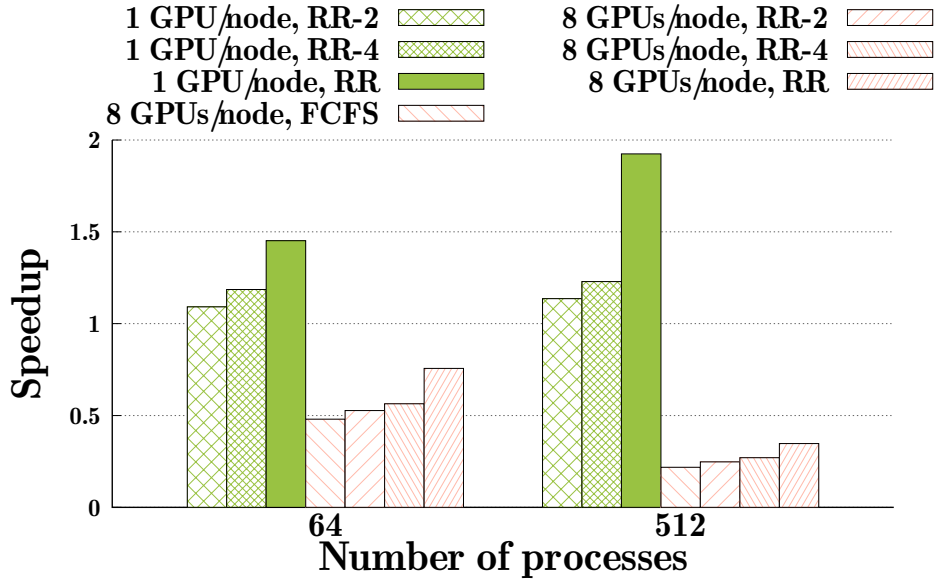

Figure 3.8: Results for Stencil4d (64 processes and 512 processes on 1D dragonfly)

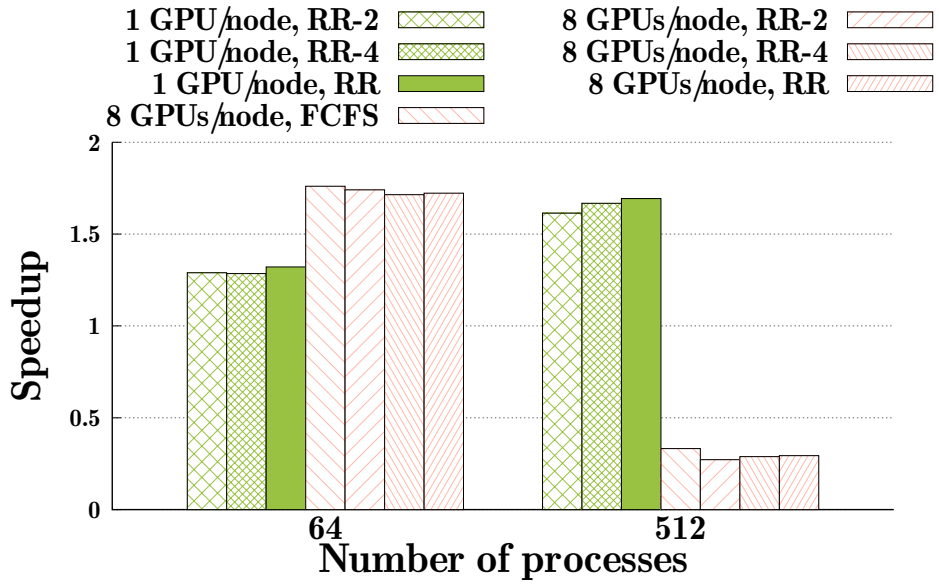

Figure 3.9: Results for Subcomm3d (64 processes and 512 processes on 1D dragonfly)

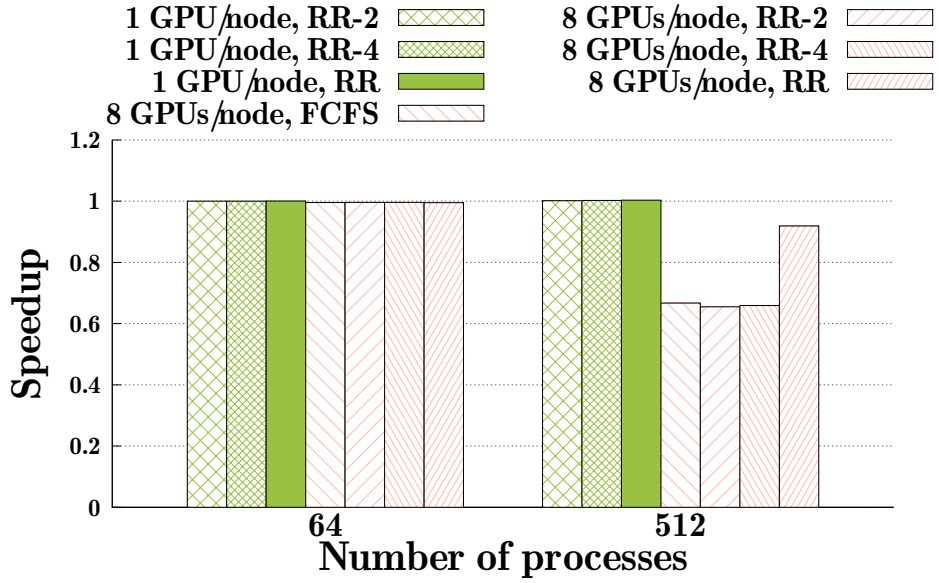

Figure 3.10: Results for Laghos (64 processes and 512 processes on 1D dragonfly)

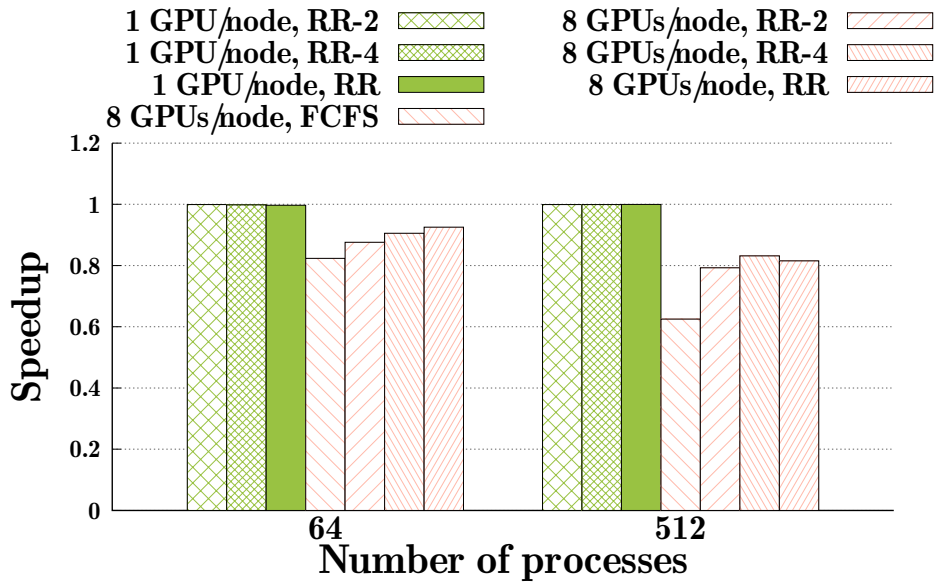

Figure 3.11: Results for SW4lite (64 processes and 512 processes on 1D dragonfly)

CHAPTER 4

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

In this work, I explore the integration of software-defined networking (SDN) into high performance computing (HPC) systems, with a focus on adapting SDN mechanisms to the distinct communication characteristics of HPC applications.

This investigation is motivated by the widespread success of SDN in domains such as data centers, campus networks, and wide-area networks, where it has demonstrated significant improvements in traffic management and network resource optimization through features like logically centralized control, per-flow management, and programmability. Prior studies, including those by Kreutz et al. and He et al., have shown how SDN can optimize Internet and data-parallel workloads through dynamic reconfiguration and intelligent traffic steering.

These efforts, while impactful, primarily target Internet and data-parallel applications characterized by coarse-grained, irregular traffic patterns, and fall short in addressing the repetitive communication phases inherent to HPC workloads. As a result, conventional SDN techniques, when applied to HPC systems, often incur prohibitive overheads due to their inability to react efficiently to short-duration, phase-driven communication behavior. Consequently, SDN remains underutilized in HPC environments, highlighting the need for novel techniques that can effectively leverage the static and predictable nature of many HPC communication patterns.

To address the challenges of applying SDN in HPC environments, I introduce techniques that leverage the communication behavior exhibited by HPC applications, specifically focusing on flow identification and communication phase identification. A large number of HPC applications simulate physical processes over numerous time steps, with each time step performing similar tasks that alternate between computation and communication phases. These communication phases often repeat and, in many applications, occur at durations in the order of subseconds or even submilliseconds, making conventional SDN techniques unsuitable due to the overheads incurred when reacting to such fine-grain behavior. However, because these communication patterns are often

static and either known to application developers or can be analyzed statically or dynamically, they present an opportunity for SDN systems to optimize their operation. I propose flow identification to accurately classify application flows, with particular focus on elephant flows—those that carry large amounts of data and typically dominate the communication time. Identifying and efficiently scheduling these elephant flows helps address the problem of unbalanced network traffic in HPC applications. Additionally, I distinguish between static and dynamic communications, using application-provided hints through an API to guide flow classification for static patterns, and employing a machine learning–based approach to support flow identification in dynamic cases.

To evaluate the effectiveness of the proposed techniques, I conduct extensive simulation experiments using the TraceR-CODES parallel discrete-event simulator on a 3-level Fat-tree topology. These simulations include both static and dynamic communication benchmarks to assess the applicability of the approach across a range of HPC workloads. For static applications such as Stencil4d, which involve near-neighbor communication patterns, the techniques I present result in notable performance improvements. For dynamic applications such as Milc, where different communication patterns are exercised across iterations, the integration of application-level hints and machine learning–based flow identification demonstrates consistent advantages over existing SDN schemes. The evaluation across all three Fat-tree topologies confirms that incorporating communication-aware strategies into the SDN framework improves performance for both predictable and unpredictable communication behaviors. These results highlight the importance of aligning SDN control mechanisms with the underlying structure of HPC communication, particularly in systems where efficient flow management directly influences application scalability.

In summary, I contribute to the ongoing effort of adapting software-defined networking to the unique requirements of high performance computing systems by exploiting the repetitive and predictable communication behaviors of HPC applications. By bridging the gap between general-purpose SDN architectures and the fine-grained, phase-driven communication patterns common in scientific computing, I offer a communication-aware control strategy that enhances flow management in HPC networks. These contributions demonstrate that combining static flow information with dynamic identification mechanisms enables SDN to support both regular and irregular communication patterns, ultimately improving performance in realistic HPC workloads and guiding future directions in adaptive network control.

4.1 Flow Classification in HPC Applications

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 methods of separating flows into elephant flows and mice flows may not be successful in HPC environments. Since, the communication characteristics of HPC application s differ greatly from those of Internet applications, to address these challenges, SHS requires a novel flow classification scheme. HPC applications often have alternating communication and computation phases with a large portion of communications being static or dynamically analyzable.

To effectively support SDN, we develop and evaluate SDN techniques that take these characteristics into consideration. Our techniques can be classified into two types, those *with no extra user information* and those *with user information*.

The flow classifier takes traffic statistics from SDN switches and performs flow classification. The difference is that our flow classifier uses a machine learning based scheme that is trained based on data from HPC workloads. Our system also allows HPC applications (SDN user) to directly give hints about their communications to the network through an API.

4.1.1 Classifying flows without user input

Threshold: Existing flow classification methods including end host-based management [48], packet sampling [44] [2], and polling per-flow statistics [49], do not assume the type of applications. In this work, we compare our proposed techniques with the classic elephant flow detection using a polling-based approach, which is based on Hedera's architecture [15]. 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 [15]. 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, we 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, we 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. We 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: Our 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: We utilized the TraceR-CODES simulator to execute representative HPC workloads, namely Random-Permutation, Shift-256, Stencil3d, Stencil4d, Milc, Nekbone, Subcom3d-a2a, Kripke [30], Laghos, SW4lite [43], and AMG [41], for ranks ranging from 32 to 512. More detailed description of these workloads is given in Section **??**. 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. We 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 4.1 shows the model prediction accuracy: the model prediction accuracy is very high for the data. We first trained the model on the applications which we are going to use for our simulations and see how it performed which are Random-Permutation, Shift-256, Stencil3d, Stencil4d, Milc, Nekbone and later on we 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. We do this to make sure that our model is capable of getting updated with new traffic as an when it comes.

Table 4.1: 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

4.1.2 Classifying flows with user input

For static communications in HPC applications like the ones in Stencil4d shown in Figure 2.4, the HPC application developer (or compiler and communication library) has the knowledge whether a communication is an elephant flow or not. If the SDN allows HPC applications (SDN users) to give hints about their communications, the flow identification task performed by the SDN can be greatly simplified. To facilitate this, we propose to have an API for SDN users to provide such information.

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 2.5 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.

4.2 Phase Identification

Flow information used to classify elephant flows in current phase should ignore the flow information for the flows in the previous phase as these two set of flows don't occur together. In order to determine if the network is in a communication phase, the network must be probed at intervals smaller than the polling interval to check if data is being sent above a certain threshold. If it is, then the network is in a communication phase; otherwise, it is mostly in a computation phase. If the network transitions from a computation phase to a communication phase or vice versa during probing, previous flow informations are disregarded, and the flow informations for each flow in the current phase is considered for prediction.

Without user input, communication phases in HPC applications can be identified using the algorithm described in Algorithm 1. The phase detection algorithim takes the polling interval, threshold for communication and number of sub intervals within a polling interval as input. The phase detection keeps on happening at every X/r intervals for the entire application run. The output of the algorithm is the current phase of the application. Initially, the algorithm starts in computation phase. At every X/r interval of time, the data sent by all leaf switches in the network are collected. This data when aggregated together represents the total communication data inside the network in that respective interval. The algorithm then checks if this aggregated data is more than the threshold for considering the network to be in communication phase. If the aggregated data is more than the threshold and the previous phase is a computation phase then there is transition and the current phase become communication. In case, the previous phase is communication phase then there is no tranistion and the application continues to be in communication phase. Whenever there is a change of phase, the flow informations for the flows in previous phase is ignored.

Algorithm 1: Phase identification algorithm

Input: Threshold p, Polling interval X, Number of sub intervals r

Output: Current phase *Current_phase*

```
Previous_phase ← computation
for each interval of X/r do
    Data_sent ← data sent by all leaf switches
    if (Data_sent > p) and (Previous_phase == computation) then
        Current_phase ← communication
    else if (Data_sent < p) and (Previous_phase == communication) then
        Current_phase ← computation
    Previous_phase ← Current_phase
```

4.3 SDN routing

In this work, I develop SDN-based routing techniques for both full-bisection fat-trees and tapered fat-trees, with support for single-path as well as multipath routing. My primary objective is to minimize congestion and balance link utilization—particularly for large elephant flows—in order to improve overall communication performance in high-performance computing systems.

Elephant flows are long-lived and bandwidth-intensive. When multiple such flows share the same network link, they contend for limited bandwidth and introduce significant congestion. This behavior makes elephant flows the primary contributors to network bottlenecks in HPC and datacenter environments. Accordingly, I define congestion as a measure of contention on a link, quantified by the number of active elephant flows traversing it.

Let the network be modeled as a directed graph $G = (V, E)$, where V is the set of switches and terminals, and E is the set of directed links. Let $F = \{f_1, f_2, \ldots, f_n\}$ denote the set of elephant flows in the network. Each flow $f \in F$ is assigned a single path $P_f \subseteq E$ by the SDN controller.

For single-path routing, the congestion $c(e)$ on a link $e \in E$ is defined as:

$$c(e) = \sum_{f \in F} \mathbf{1}_{\{e \in P_f\}}$$

Here, $\mathbf{1}_{\{e \in P_f\}}$ is an indicator function that equals 1 if link e is part of the path assigned to flow f, and 0 otherwise.

The maximum link congestion across the network is then given by:

$$C_{\max} = \max_{e \in E} c(e)$$

This congestion metric forms the basis for evaluating routing efficiency in my single-path SDN scheme. It guides the routing decision process by identifying and minimizing the worst-case contention experienced on any single link in the network.

4.3.1 SDN-Singlepath Routing

In single-path SDN routing, I assign each elephant flow a unique route through the network. Unlike traditional static routing methods, which rely on preconfigured tables, SDN enables dynamic path selection using a global view of current link utilization. This centralized perspective allows the controller to make informed decisions that minimize congestion and avoid oversubscription on critical links.

To explore the design space of single-path routing under SDN, I implement two complementary strategies: *SDN-greedy* and *SDN-optimal* which we talk about in the following sections.

SDN-greedy. *SDN-greedy* is a lightweight, single-path routing algorithm designed to operate on both full-bisection and tapered fat-tree topologies. The algorithm leverages the SDN controller's global view of the network to make path selection decisions that minimize link congestion in real time.

As shown in algorithm 2, the controller processes each elephant flow individually. For a given flow, it enumerates all feasible paths between the source and destination switches. For each candidate path, it computes the maximum link congestion $c(e)$ across all links e on the path, which corresponds to the path's worst-case contention. This value is equivalent to evaluating the local $C_{\max}$ for that path. The controller then selects the path with the smallest such value and assigns it 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 to reduce the risk of bottlenecks. Due to its simplicity and responsiveness, SDN-greedy is well-suited for environments with light to moderate traffic.

However, because flows are routed independently, SDN-greedy may cause multiple flows to converge on the same links, especially under heavy or bursty traffic. This lack of global coordination can lead to suboptimal load balancing in dense communication scenarios.

Algorithm 2: SDN-greedy algorithm

```
Input: Elephant flows in a phase EF
Output: Load-balanced routing table Routing_table
Function ComputeLoadBalancedRoutingTable(EF):
    Initialize an empty map Routing_table
    foreach E ∈ EF do
        Get current link loads in the network
        Get all possible paths for Source and Destination of E
        foreach path in All_possible_paths do
            Compute maximum link congestion c(e) for links in path
        Add path with minimum C_max to Routing_table for E
    return Routing_table
```

4.3.2 SDN-Optimal Routing Algorithm

In fat-tree topologies, particularly full-bisection fat-trees, minimizing link contention is crucial to improving communication performance. In this work, I introduce a single-path routing algorithm, called *SDN-optimal*, that achieves the theoretically lowest possible value of maximum link congestion, denoted by $C_{\max}$, across all network links. Specifically, SDN-optimal guarantees that $C_{\max} = 1$ for any permutation of elephant flows in a full-bisection fat-tree. This is achieved by (1) partitioning the flows into a minimum number of disjoint permutations, and (2) scheduling each permutation such that no two flows share any link. The second step leverages the structural properties of full-bisection fat-trees and Hall's Marriage Theorem to ensure contention-free routing.

SDN-optimal Algorithm for Full-Bisection Fat-Trees. The SDN-optimal algorithm consists of two main steps: (1) partitioning the traffic pattern into k disjoint permutations, and (2) routing each permutation using a contention-free schedule.

Step 1: Flow Partitioning into Permutations.. Given a set of elephant flows F, I construct a bipartite graph $G = (S, D, E)$, where S and D are source and destination nodes, and each edge $e \in E$ represents a flow. To ensure that each source and destination has the same number of flows (i.e., a regular graph), I add dummy nodes and edges to make the graph k-regular, where k is the maximum degree in the original graph. I then apply edge-coloring (via Kőnig's Theorem) to decompose the

graph into k disjoint perfect matchings. Each matching corresponds to a permutation $P_1, P_2, ..., P_k$, where each node appears exactly once per permutation. The algorithm is described in algorithm 3

Algorithm 3: Flow Partitioning into Disjoint Permutations

Input: Flow set F, max degree k
Output: k disjoint permutations $P_1, ..., P_k$
Construct bipartite graph $G = (S, D, E)$ from flows in F
Pad G with dummy nodes/edges to make it k-regular
Apply edge-coloring to obtain k disjoint matchings $P_1, ..., P_k$
Remove dummy flows from each P_i
return $P_1, ..., P_k$

Step 2: Contention-Free Scheduling.. For each permutation P_i, the flows are routed through the fat-tree using link-disjoint paths. In a full-bisection bandwidth of the topology, because the number of flows per permutation equals the number of available links at each network layer, Hall's Marriage Theorem guarantees the existence of a perfect matching, no link will be used by more than one flow in a given permutation. Therefore, the maximum congestion per link in each permutation is

exactly 1. The algorithm is described in algorithm 4

Algorithm 4: Contention-Free Scheduling for Permutation P_i

Input: Permutation P_i, topology G, link usage table L
Output: Routing paths for all flows in P_i

```
// Step 1: Schedule Intra-Pod Flows
foreach intra-pod flow f ∈ P_i do
    foreach uplink u from source leaf to aggregation do
        if u is unused then
            Find corresponding downlink d to destination leaf
            if d is unused then
                Assign path: leaf → agg → leaf
                Mark u, d as used; break
    if no path assigned then
        Apply non-blocking rearrangement to free a valid intra-pod path
// Step 2: Schedule Inter-Pod Flows
foreach inter-pod flow f ∈ P_i do
    foreach uplink u_1 from source leaf to aggregation do
        if u_1 is unused then
            foreach uplink u_2 from aggregation to core do
                if u_2 is unused then
                    Find downlink d_2 from core to destination agg
                    if d_2 is unused then
                        Find downlink d_1 to destination leaf
                        if d_1 is unused then
                            Assign full path: leaf → agg → core → agg → leaf
                            Mark all links as used; break
    if no path assigned then
        Apply non-blocking rearrangement to free a valid inter-pod path
```

Proof Sketch of Optimality.. In a full-bisection fat-tree, the number of aggregation switches is equal to the number of leaf switches. Each leaf switch has $\frac{K}{2}$ uplinks to aggregation switches. The total number of links between the leaf and aggregation switches is $N_{\text{leaf}} \times \frac{K}{2}$. Since each compute node connects to a downlink port on a leaf switch, the total number of compute nodes is also $N_{\text{leaf}} \times \frac{K}{2}$. In a full permutation traffic pattern, each compute node sends and receives exactly one flow, so the total number of flows in each permutation is equal to the number of compute nodes. Therefore, the number of flows in each permutation is equal to the number of links between the leaf and aggregation switches.

Similarly, the number of links between the aggregation and core switches is also equal to the

number of flows in each permutation, because each aggregation switch has $\frac{K}{2}$ uplinks to core switches. As a result, the number of flows in each permutation exactly matches the number of available links at each layer of the network.

This one-to-one correspondence between flows and available links allows the use of Hall's Marriage Theorem to guarantee a contention-free assignment. Suppose, for contradiction, that in some permutation, a link is used by more than one flow. Then at least one other link must remain unused, violating the condition required for a perfect matching. This contradiction implies that no two flows in the same permutation can share a link. Therefore, the maximum congestion on any link in any permutation is 1. Since this holds for every permutation, the maximum link congestion across all permutations is also 1. Thus, the SDN-optimal algorithm guarantees $C_{\max} = 1$ and achieves contention-free routing in a full-bisection fat-tree.

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, flows begin to contend for shared links, and the assumptions of perfect matching guaranteed by Hall's Marriage Theorem no longer hold. Consequently, achieving a contention-free assignment for all flows within a single permutation becomes infeasible.

To overcome this limitation and preserve optimal scheduling, 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.

Our goal is to make sure that we create sub-permuatations by selects 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 to be applied to guarantee a conflict-free routing for each sub-permutation.

Algorithm 5: Sub-Permutation Construction Using Core and Aggregate Link Counters

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

```
1  P_list ← ∅;
2  while F_core ≠ ∅ or F_agg ≠ ∅ do
3      P ← ∅;
4      core_links ← c;
5      agg_links ← a;
       // Stage 1:  Add core-level flows
6      foreach f ∈ F_core do
7          Add f to P;
8          core_links ← core_links − 1;
9          Remove f from F_core;
10         if core_links == 0 or F_core == ∅ then
11             break
       // Stage 2:  Add aggregate-level flows
12     foreach f ∈ F_agg do
13         Add f to P;
14         agg_links ← agg_links − 1;
15         Remove f from F_agg;
16         if agg_links == 0 or F_agg == ∅ then
17             break
18     Append P to P_list;
19 return P_list;
```

4.3.3 SDN-Multipath Routing

Multipath routing enables traffic to be split across multiple paths, improving bandwidth utilization and often reducing congestion. However, in many scenarios where flow paths overlap significantly, for example in shift traffic, this can introduce additional contention. In contrast, single-path routing (such as SDN-optimal) can perform better when flows are carefully scheduled to avoid bottlenecks in those cases.

To intelligently decide between these two strategies, I propose *SDN-adaptive* routing for the multipath routing.

SDN-adaptive. The key idea behind SDN-adaptive is to first gather runtime information about the application's traffic characteristics. During an initial profiling phase, SDN-adaptive collects communication performance data by running one iteration using adaptive multipath routing. This information is then sent to the SDN controller, which uses it decide the routing. The process begins by executing one iteration of the application using adaptive multipath routing. The SDN controller records the communication time from this iteration as a reference. Then, the full simulation is restarted using SDN-optimal routing. After completing the first iteration of this SDN-optimal run, the controller compares the current communication time with the previously recorded time from the adaptive run.

If SDN-optimal demonstrates better performance (i.e., lower communication time), it is retained for the rest of the simulation. If, however, SDN-optimal is slower than adaptive, this indicates that multipath routing is more effective for the application's communication pattern. In that case, the controller immediately switches to adaptive routing for the remainder of the simulation.

This design allows the routing policy to be tuned to the observed behavior of the application, ensuring better adaptability across diverse workloads. The algorithm is described in algorithm 6

Algorithm 6: SDN-adaptive algorithm

Input: Application A;
Topology G;
Routing schemes: SDN-optimal and Adaptive
Output: Final routing strategy and flow configuration

```
Run one iteration of A using Adaptive routing
Record commTime_Adaptive
Start full simulation of A using SDN-optimal routing
After first iteration, record commTime_SDN-optimal
if commTime_SDN-optimal < commTime_Adaptive then
    Continue simulation with SDN-optimal routing
else
    Switch to Adaptive routing for remainder of simulation
```

4.4 Experimental Setup

I perform extensive experiments using the TraceR-CODES simulator [21, 35], which I have extended to support the study of the SDN techniques discussed in this work. 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.

To support my experiments, I have integrated the trained DNN model into the simulator using Google TensorFlow's C APIs. To ensure compatibility with the TensorFlow libraries, I carefully selected the MPI and GNU compiler standards. In particular, I utilized MVAPICH 2 as the MPI implementation and adopted C++14 from GNU version 9.0.1 as the compiler standard.

By leveraging real-time network statistics, my DNN model dynamically predicts network flows during the simulation runtime. This capability not only enhances prediction accuracy but also enables real-time data processing, making the model both robust and efficient.

In addition to evaluating SDN-based routing, I also investigate the impact of hardware design characteristics on performance using the Fat-tree connection topology. The Fat-tree topology is a well-established interconnect architecture commonly employed in HPC systems and data centers. I conduct simulations on two distinct Fat-tree configurations: a 1024-node full-bisection Fat-tree and a 1536-node tapered Fat-tree with a 3:1 oversubscription ratio. In both configurations, each switch is equipped with 32 ports, and each leaf switch connects to 16 compute nodes. Importantly, 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.

In this study, I explore three types of SDN routing mechanisms:

SDN-greedy, which allocates paths by minimizing congestion in real time using simple heuristics;

SDN-optimal, which computes globally least-congested paths by evaluating all possible routes;

SDN-adaptive, which dynamically adjusts routing decisions based on the current traffic patterns and flow types.

For flow classification, I employ three distinct strategies:

Threshold-based, where flows are classified as elephant or mice based on a predefined data volume threshold;

DNN-based, where a trained deep neural network identifies flow types based on extracted runtime features;

User input, where the application provides direct hints to classify communication flows.

Each of these methods is described in detail in earlier sections of this work. Below is the table outlining the network configuration parameters used in the simulation.

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

Table 4.2: 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

us to eliminate any extraneous delay factors and isolate the effects of the communication process on our data analysis.

4.5 Application and Workloads

To evaluate the effectiveness of SDN-based routing strategies under varying communication patterns, I select a set of communication-intensive applications. These applications have been described in detail in Chapter 2, including their computation and communication characteristics, as well as their relevance to realistic HPC workloads.

In this study, I use six representative applications: `Random-Permutation`, `Shift`, `Stencil3d`, `Stencil4d`, `Milc`, and `Nekbone`. 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.

To further investigate the impact of phase-driven communication behavior, I also design two synthetic mixed-traffic applications:

- **Random-Permutation-Mixed**: This pattern alternates between two distinct random-permutation communication phases, separated by a computation phase. Each communication phase uses a different source-destination mapping, 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. The transition between patterns allows evaluation of the routing system's responsiveness to changes in spatial communication demands.

To ensure accurate analysis of communication transitions, 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.

4.6 Performance Study

The result of various SDN techniques which are used in HPC is presented here

4.6.1 Evaluation of flow classification techniques

In this section, we compare the impact of different flow identification techniques on the SDN algorithm for various applications under full bisection fat-tree and 3-to-1 tapered fat-tree configurations. The evaluation includes performance metrics across application communication time for different applications.

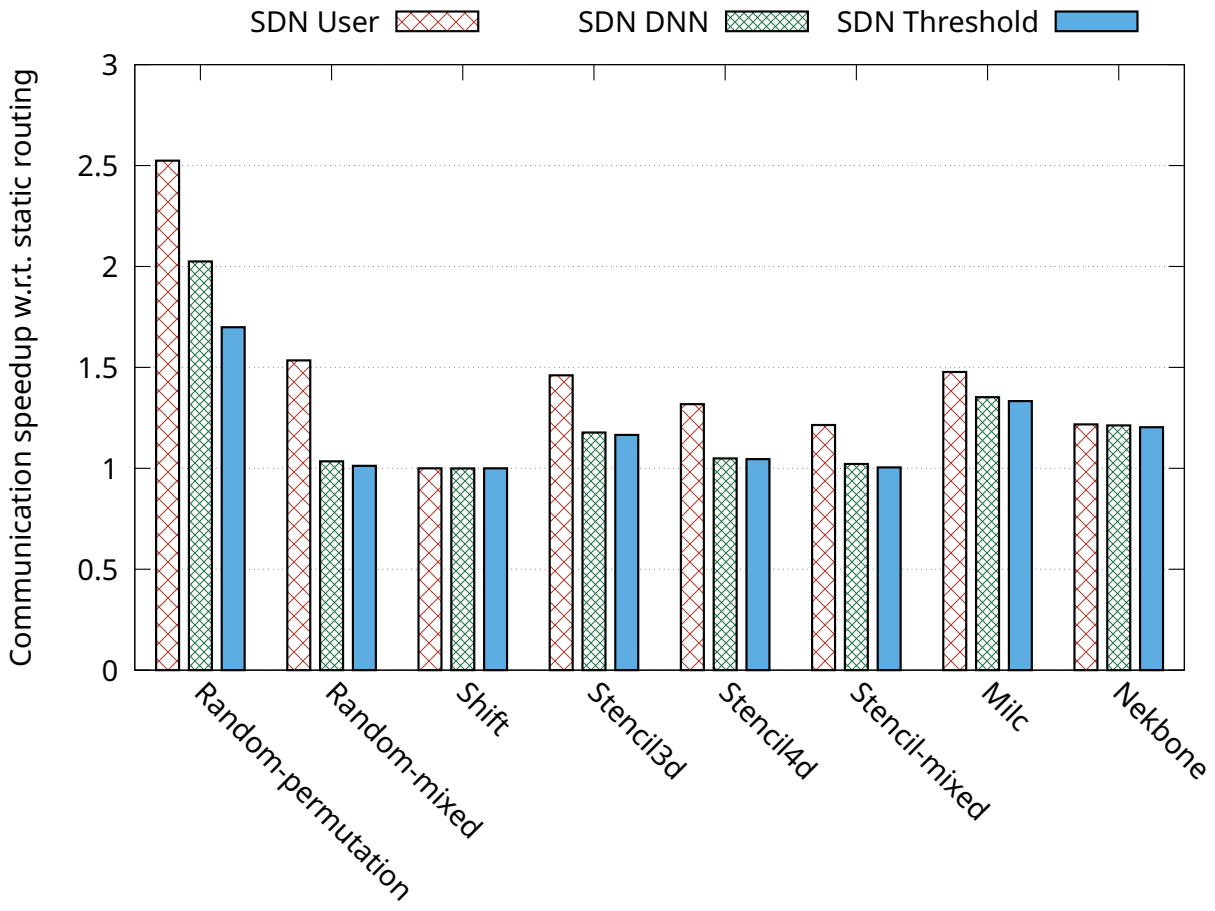

Figure 4.1: Comparison of flow detection in full bisection fat-tree of 1024 nodes

Performance Under Full Fat-Tree. The figure 4.1 compares communication speedup achieved by three flow detection techniques—SDN User, SDN DNN, and SDN Threshold—across various applications, including Random-Permutation, Random-mixed Shift, Stencil3D, Stencil4D, Stencil-mixed, Milc, and Nekbone. These applications represent diverse traffic patterns, ranging from simple to highly complex workloads, making them ideal benchmarks for evaluating network performance. A higher bar (speedup) signifies better efficiency and lower communication time relative to the baseline. The user flow identification technique performs best due to early flow identification, enabling prompt traffic management. DNN detection ranks second, limited by a 0.3-millisecond delay. Threshold detection performs worst due to delayed flow recognition. Importantly, all three techniques perform better than the widely used single-path static routing, demonstrating the effectiveness of dynamic flow detection in improving communication efficiency within SDN frameworks. User detection consistently leads across all applications. DNN detection shows reasonable gains despite initial delay. Threshold detection offers minimal improvement. These results highlight the importance of incorporating flow detection mechanisms to enhance SDN routing performance beyond conventional static routing methods like D-mod-K, also, the DNN model does a faster flow classification compared to threshold based model without losing in performance.

Performance Under Tapered Fat-Tree. The figure 4.2 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 tapered fat-tree topology, even a random-permutation pattern represents a relatively dense communication workload. To better demonstrate the effectiveness of our routing mechanism in balancing traffic, we 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. Our evaluation revealed that SDN User consistently achieved the highest speedup, exceeding 6x, 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. Similar trends were observed in the Shift application, where SDN User maintained its superior performance, while SDN DNN and SDN Threshold provided moderate improvements over static routing. In compute-heavy applications such as Milc and Nekbone, the

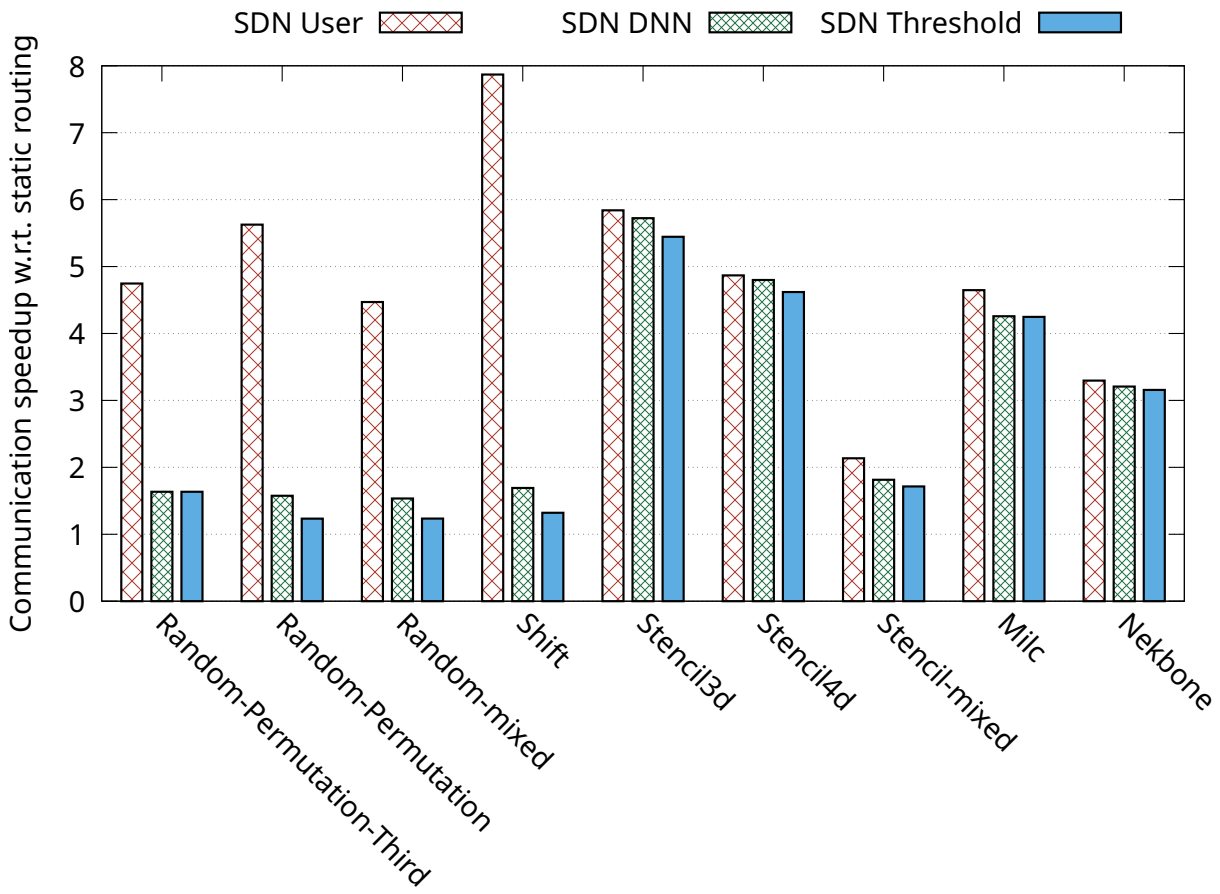

Figure 4.2: Comparison of flow detection in 3 to 1 taper fat-tree of 1536 nodes

performance gap between techniques narrowed, though SDN User remained the top performer. The results suggest that compute-heavy traffic benefits from steady-state optimizations, though early flow detection consistently outperformed or matched the threshold-based approach. These findings underscore the critical role of early flow detection in achieving optimal traffic balancing, particularly in tapered Fat-Tree topologies with constrained bandwidth, as reflected by higher speedup values relative to baseline D-mod-K routing.

4.6.2 Evaluation of Phase Identification

This section evaluates the effectiveness of phase identification, across varioous applications under full bisection fat-tree and 3-to-1 tapered fat-tree configurations.

Performance Under Full Fat-Tree. The figure 4.3 shows phase identification results for full Fat-tree, where the x-axis represents different workloads, including Random-permutation, Random-

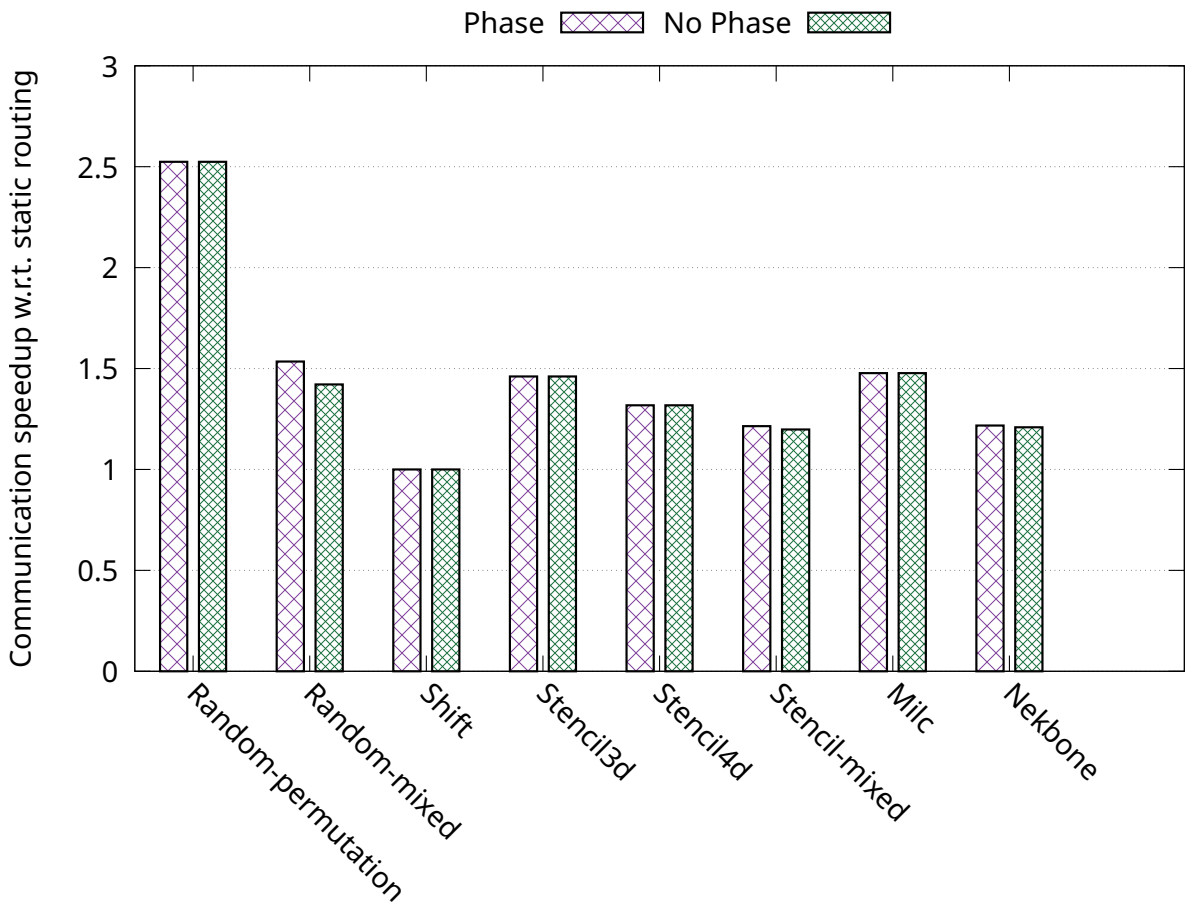

Figure 4.3: Comparison of phase identification in full bisection fat-tree of 1024 nodes

mixed, Shift, Stencil3d, Stencil4d, Stencil-mixed, Milc, and Nekbone, while the y-axis quantifies the relative speedup. In Full Fat-Tree routing, phase-based optimization leverages threshold-based flow classification and SDN routing to improve communication efficiency. The analysis of communication speedup reveals notable improvements in random-mixed, stencil-mixed, and nekbone, with random-mixed achieving a 7.98% increase in performance. The phase identification mechanism efficiently distinguishes between different communication phases. When a phase transition occurs, it promptly loads the corresponding routing table, ensuring seamless adaptation to dynamic communication patterns.

The evaluation time for each phase is one-tenth of a polling phase, during which the phase identifier continuously monitors network flows to detect transitions between computation and communication phases. Upon detecting a phase change, it swiftly loads the appropriate routing table to optimize data flow. In random-mixed, stencil-mixed, and nekbone, frequent transitions between

computation and communication phases trigger rapid routing table updates, leading to measurable performance improvements in these applications.

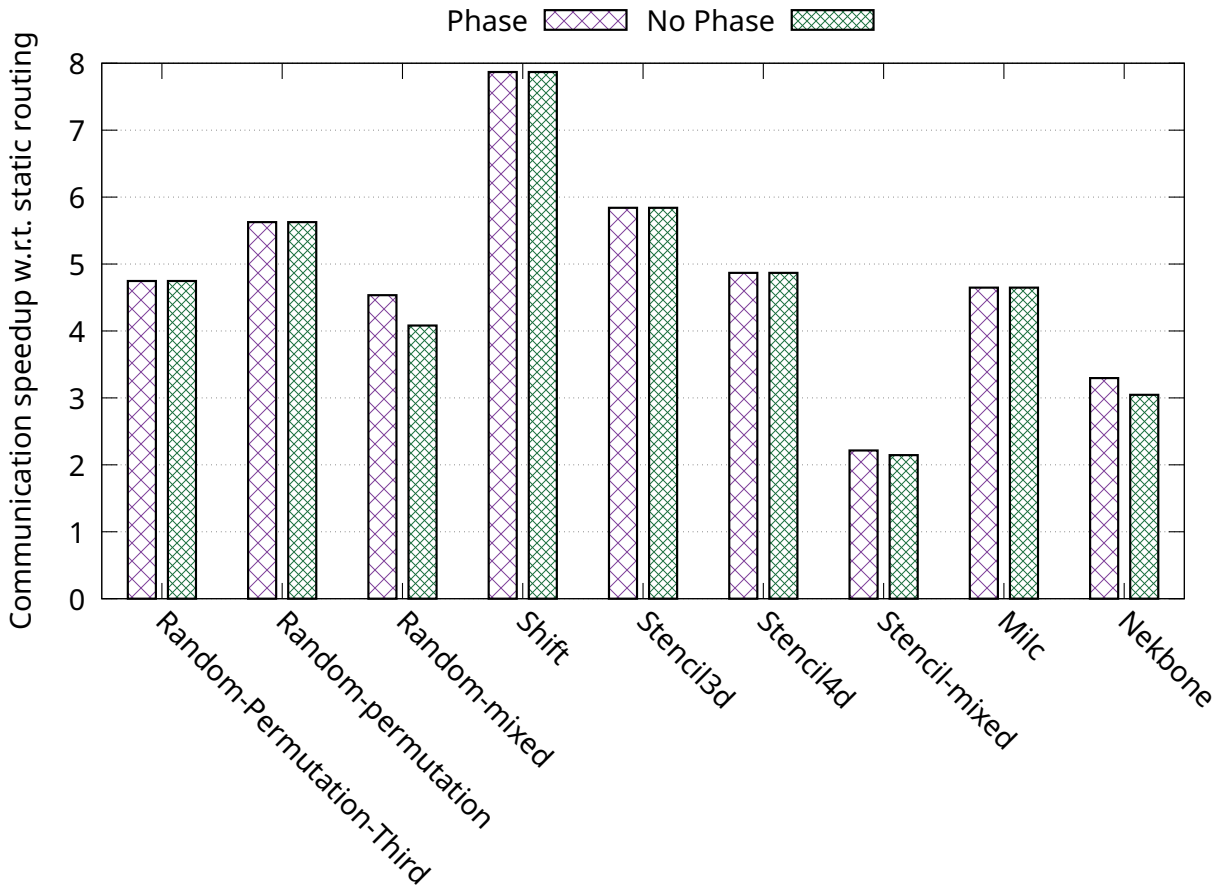

Figure 4.4: Comparison of phase identification in full bisection fat-tree of 1536 nodes

Performance Under Tapered Fat-Tree. The figure 4.4 shows phase identification results for full Fat-tree, In Tapered Fat-Tree routing, phase-based optimization dynamically adjusts traffic injection rates using threshold-based flow classification and SDN routing to further enhance communication efficiency. The analysis of communication speedup shows notable improvements in random-mixed, stencil-mixed, and nekbone, with random-mixed achieving an 11.13% increase in performance, surpassing the gains observed in Full Fat-Tree routing. The phase identification mechanism efficiently detects phase transitions and dynamically manages traffic flow to reduce network congestion

4.6.3 Evaluation of SDN-based Routing

This section evaluates the effectiveness of SDN-based routing approaches across various applications under full bisection fat-tree and 3-to-1 tapered fat-tree configurations of user identification of flows. We analyze performance in terms of communication time.

Performance Under Full Fat-Tree. The figure 4.5 illustrates the communication speedup achieved by three routing techniques—Adaptive, SDN User, and SDN-Adaptive across various applications under a full fat-tree topology. The y-axis represents the speedup relative to static D-mod-K routing, where higher bars indicate better performance.

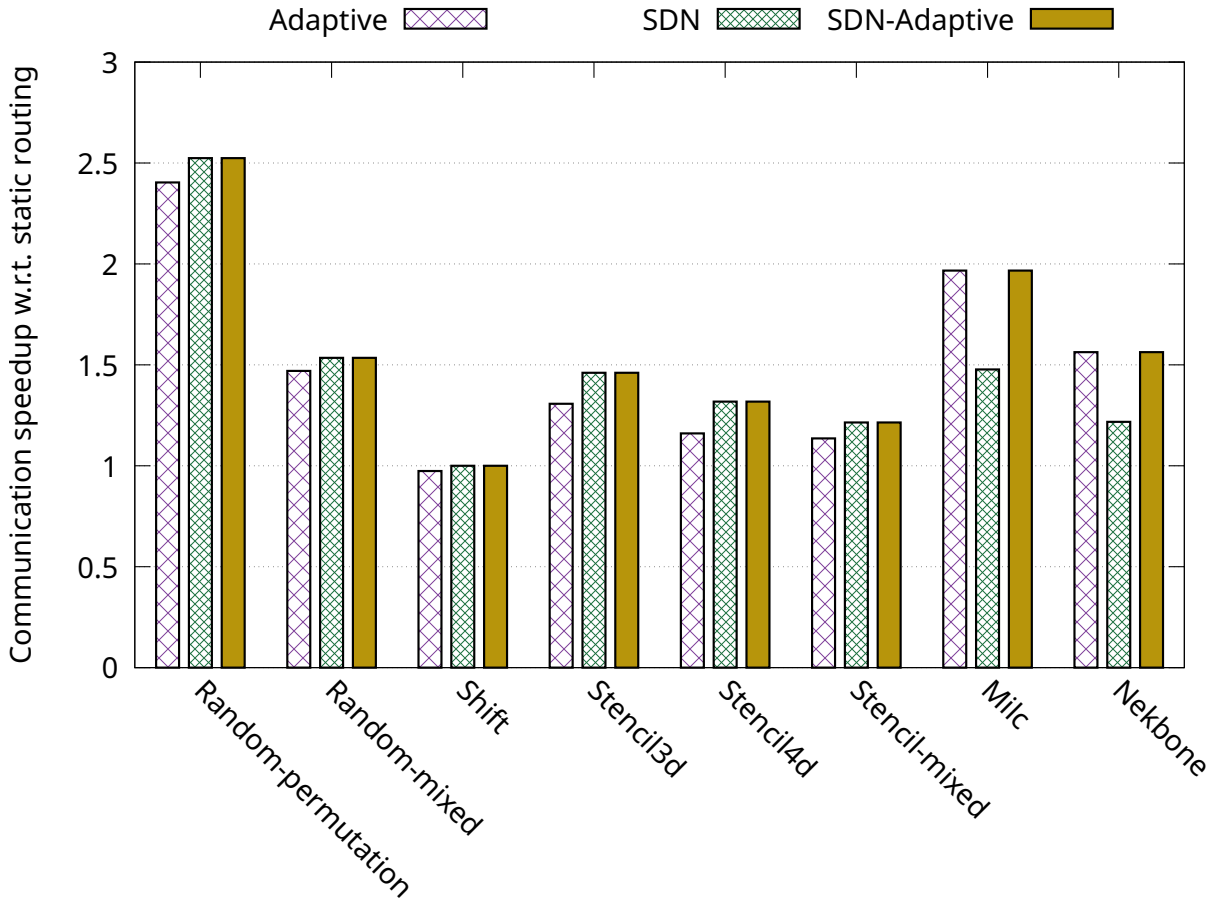

Figure 4.5: Comparison of routing techniques in full fat-tree of 1024 nodes

Single-path SDN routing consistently outperforms single-path DmodK routing across all scenarios.

While SDN routing generally performs better than Adaptive routing, the multipath nature of Adaptive routing allows it to balance the load more effectively when communication becomes dense. At this point, SDN-Adaptive, a hybrid approach combining SDN and Adaptive strategies, performs similarly to Adaptive routing at high communication density application and similar to SDN then the communication density is low, leveraging its flexibility to adapt based on the application's requirements.

SDN even being a single path routing is kind of making sure that flows are completing together and there is no stragglers left behind.

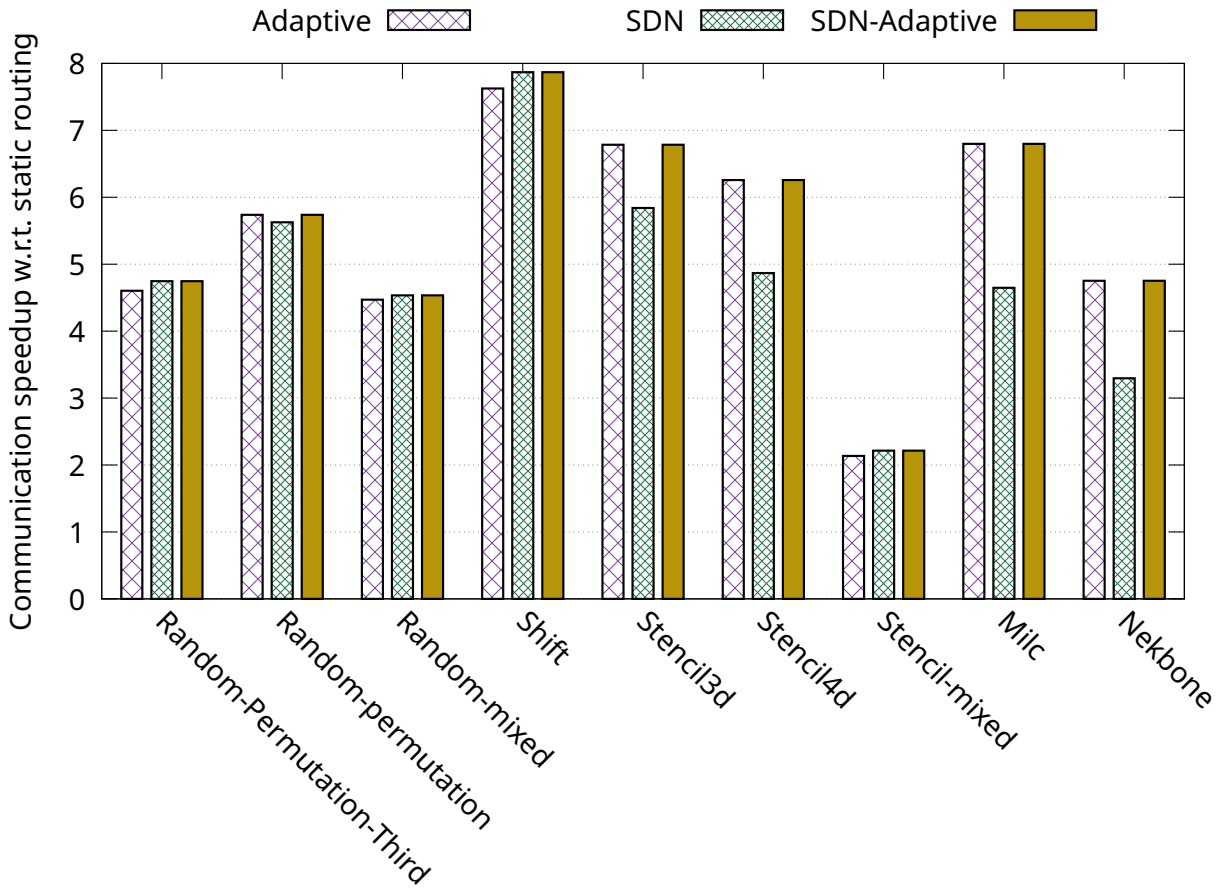

Figure 4.6: Comparison of routing techniques in 3 to 1 taper fat-tree of 1536 nodes

Performance Under Tapered Fat-Tree. The figure 4.6 presents a comparison of routing techniques (Adaptive, SDN User, and SDN-Adaptive Use) across three applications (Random-Permutation, Shift, and Milc) under a Tapered Fat-Tree topology. The y-axis represents the com-

munication speedup relative to DmodK routing, where higher bars indicate better performance. In a Tapered Fat-Tree topology, applications typically experience dense traffic patterns due to limited bandwidth at higher levels caused by tapering. Adaptive routing demonstrates strong performance in balancing network load, especially under traffic-heavy scenarios. However, in Random-Permutation-Third, where traffic is reduced, our evaluation clearly shows that SDN-based routing outperforms adaptive routing. In Shift traffic patterns, where traffic predictability is higher, Adaptive routing effectively balances the load better than static routing, though SDN-based routing still makes slightly better decisions due to its global network awareness. SDN-Adaptive routing leverages the strengths of both Adaptive and SDN-based routing by dynamically adapting to application-specific traffic patterns, selecting the most suitable approach for optimal performance.

4.7 Summary

This chapter explores the adaptation of software-defined networking (SDN) techniques to the distinct communication patterns of high performance computing (HPC) applications. While SDN has been successful in traditional data center and wide-area network environments, its use in HPC systems has remained limited due to challenges posed by fine-grained and phase-driven communication behavior. This study addresses those limitations by introducing novel flow classification and communication phase identification mechanisms specifically designed to align with HPC workloads.

The core of this work involves a detailed simulation-based evaluation using the TraceR-CODES framework. Three flow classification approaches are considered: threshold-based classification, deep neural network (DNN)-based classification, and user-input-based classification. These are applied in conjunction with three SDN routing strategies: greedy routing, optimal routing, and adaptive routing. The techniques are evaluated across real and synthetic HPC workloads using two common network topologies: a 1024-node full-bisection fat-tree and a 1536-node 3-to-1 tapered fat-tree.

The results show that the strategies implemented in SDN routing perform very well under single-path routing. They significantly reduce network congestion and ensure the efficient execution of HPC applications within SDN-enabled systems. The multipath variant, SDN-adaptive, dynamically selects between SDN and adaptive routing and achieves performance that is either better than or comparable to adaptive routing alone, depending on the type of application and the density of communication. Furthermore, the use of phase-aware routing and early flow identification, particularly through user input or DNN-based models, plays a key role in adapting routing behavior to dynamic traffic patterns and improving overall communication efficiency.

By aligning SDN's programmability and global control with the communication characteristics of HPC workloads, this work demonstrates that SDN can serve as a robust and efficient networking solution for modern supercomputing platforms. These contributions support the broader integration of SDN in HPC systems and provide a solid foundation for future research in intelligent, adaptive network control.

CHAPTER 5

CONCLUSION

In my dissertation research, I focus on two critical areas in High-Performance Computing (HPC): optimizing hardware parameters for next-generation GPU-based platforms and integrating Software Defined Networking (SDN) to enhance HPC application performance.

In the first 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 analyze 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 the second part, I investigate routing optimization strategies for HPC networks using software-defined networking (SDN). Specifically, I develop a suite of SDN-based routing algorithms—adaptive, optimal, and a hybrid scheme tailored for tapered fat-tree topologies—to address performance limitations caused by flow-level contention. The SDN-optimal algorithm ensures contention-free routing in full-bisection fat-trees by leveraging a graph-theoretic formulation and Hall's Marriage Theorem. For tapered fat-trees, where upper-layer bandwidth is reduced, I introduce a permutation-splitting strategy to construct sub-permutations that maintain link-level contention bounds. Additionally, I propose an SDN-adaptive approach that selects between adaptive and optimal routing modes dynamically based on runtime flow behavior. Simulation results demonstrate that these routing strategies reduce congestion and improve communication performance in an application-dependent manner, highlighting the importance of topology-aware, programmable routing in next-generation HPC systems.

In summary, my dissertation research contributes to optimizing HPC platforms by addressing both hardware and networking challenges. By enhancing GPU integration and utilizing SDN 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] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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`.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

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

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

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

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

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

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

[35] 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.

[36] 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.

[37] Sabine R Ohring et al. "On generalized fat trees." In: *Proceedings of 9th international parallel processing symposium*. IEEE. 1995, pp. 37–44.

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

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

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

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

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

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.