FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

EVALUATING NETWORK PARAMETERS AND SDN STRATEGIES FOR ENHANCING PERFORMANCE IN HPC APPLICATIONS

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

2024

Saptarshi Bhowmik defended this thesis on May 5, 2024.
The members of the supervisory committee were:

Xin Yuan
Professor Directing Thesis

Fengfeng Ke
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 rise in demand for High-Performance Computing (HPC) applications, driven by complex scientific computations, poses challenges in optimizing communication performance, especially in fat-tree-based cloud data centers. This prospectus proposes integrating Software-Defined Networking (SDN) enhancements into fat-tree topologies to enhance communication efficiency for HPC applications. By alleviating communication bottlenecks, this research aims to facilitate seamless execution of HPC workloads on cloud platforms, thereby enabling groundbreaking discoveries in scientific computing. Through a comprehensive methodology, including simulation validation and implementation of SDN routing strategies, this study seeks to uncover insights into optimizing communication efficiency in next-generation HPC systems. The findings are anticipated to extend existing knowledge and influence the design of future supercomputers, paving the way for enhanced performance in cloud-based HPC environments.

CHAPTER 1

INTRODUCTION

In recent years, the demand for High-Performance Computing (HPC) applications has surged, driven by the increasing complexity of scientific computations and data-intensive tasks. Despite the remarkable progress in HPC infrastructure, challenges persist in optimizing communication performance, particularly in modern HPC environment like cloud data centers. This prospectus aims to address these challenges by incorporating Software-Defined Networking (SDN) [26] enhancements for optimizing routing for these state of the art HPC environments, thereby improving the efficiency and reliability of communication for HPC applications.

Cloud data centers, such as those offered by GCP and Azure, heavily rely on fat-tree topologies [27] for their network architecture [11]. However, the bottleneck in these systems often lies in communication efficiency, hindering the seamless execution of HPC applications. Between 2010 and 2018, there was a remarkable 65-fold surge in the computational throughput of the Top 500 HPC systems [1], whereas the increase in offnode communication bandwidth was comparatively modest, standing at only 4.8 times [8] [32]. By enhancing the routing techniques within fat-tree topologies through SDN, this research seeks to alleviate these bottlenecks and facilitate the deployment of HPC workloads on cloud platforms. Such improvements hold significant implications for scientific computing, enabling researchers to harness the full potential of HPC resources for groundbreaking discoveries across various fields.

The research methodology entails a comprehensive approach to maximize the performance of fat-tree topologies with SDN enhancements.

Initially, a thorough validation of the simulation model against real-world HPC systems, such as Quartz [29], will be conducted to understand the effects of network parameters (e.g., link bandwidth, NIC scheduling) on communication performance.

Subsequently, three distinct SDN routing strategies will be implemented and evaluated: greedy single-path routing, multipath routing, and heuristic-based optimal routing. These strategies will be tested within a full bisection fat-tree topology to assess their efficacy in improving communication performance for HPC applications.

The primary contribution of this research lies in its exploration of how network parameters impact communication in next-generation HPC systems. By integrating SDN enhancements into fat-tree topologies, novel insights will be gained into optimizing communication efficiency for data-centric applications. Furthermore, this work will extend existing knowledge by demonstrating the potential of SDN-driven routing techniques to significantly enhance the performance of HPC workloads in cloud environments.

The structure of this prospectus aligns with the outlined research objectives. Chapter 2 will provide a comprehensive overview of interconnection technologies and SDN fundamentals. In Chapter 3, the focus will shift to examining 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 and insights derived from the research findings.

CHAPTER 2

BACKGROUND

In the domain of High-Performance Computing (HPC) and data center networks, the coordination of numerous hardware components is crucial for them to function as a unified system. This coordination happens through an interconnection network, which serves as the backbone for communication among these components. Thousands of hardware pieces collaborate over this interconnection network to ensure smooth operation. The effectiveness of this interconnect relies on various design choices, such as the topology used to connect physical components, the routing scheme to select communication paths, and managing network traffic loads along these paths and links. In this chapter, I provide essential background information on topology, routing and Software Defined Networks (SDN). 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. We usually refer to servers as processing elements, and switches or routers as forwarding elements. When two points are directly connected by a line, we 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, we want a low diameter and high bisection bandwidth to perform well at scale. We 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 [27]. 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.

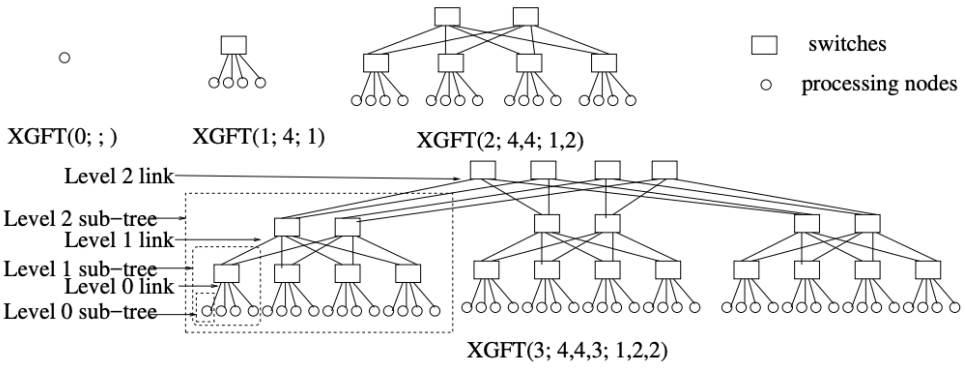

Figure 2.1: A fat-tree represented in XGFT.

Fat-tree topology ensures that the bandwidth within a level remains consistent, if not increasing, as one moves toward higher levels[27, 21]. 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[27]. 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[33]. 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.

2.1.2 Dragonfly

The Dragonfly topology stands out as a cost-effective solution for building expansive interconnection networks [24]. This design is characterized by its two-layer structure, exemplified in Figure 2.2. Initially proposed by Kim et al. [24], 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 2.2, 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 [17]. 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) [24].

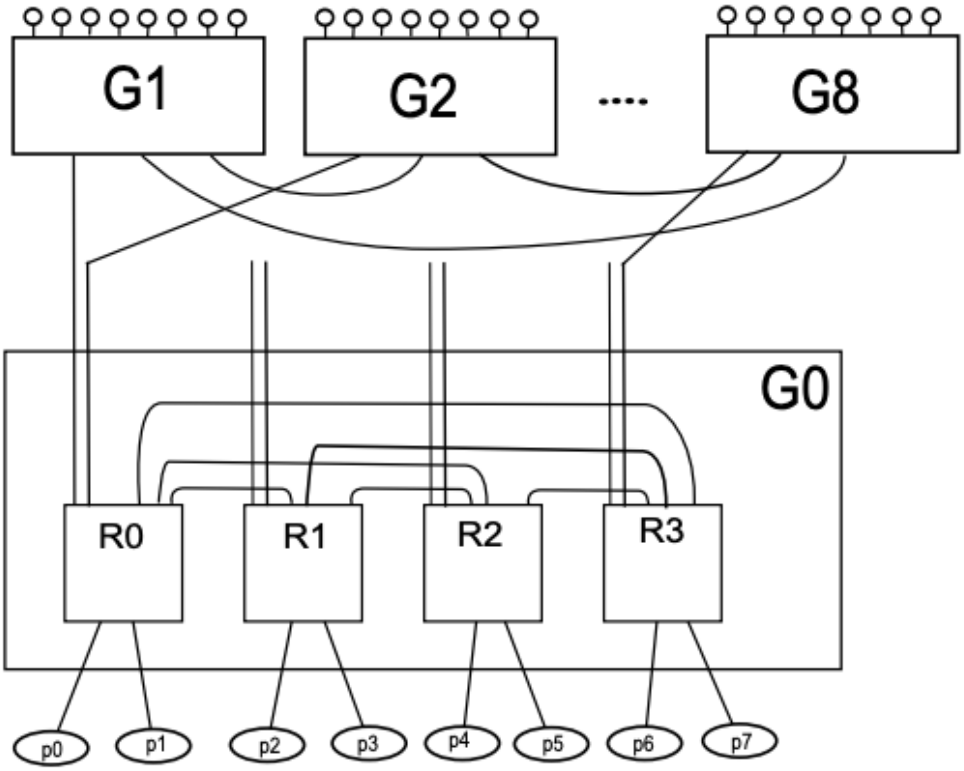

Figure 2.2: Dragonfly architecture with 9 groups and 4 routers per group

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 [12, 10]. These implementations have found their way into current supercomputers like Titan [34] and Trinity [6], 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, we have the Minimal Routing (MIN) scheme [24], 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 we 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, we have Valiant Load-Balanced Routing (VLB) [24, 22], 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, we have Universal Globally Adaptive Load-Balanced Routing (UGAL) [24, 22], 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. The following is a brief description of the applications we used in our work:

- **Random permutation**: A synthetic traffic where each node sends 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 [28].
- **Laghos**: Proxy application that solves time-dependent Euler equations with MPI-based domain decomposition [30].
- **AMG**: Parallel algebraic multigrid solver [35].
- **SW4lite**: Proxy application for SW4 [36], a 3D seismic modeling code.
- **Lammps**: LAMMPS is an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator, it equations of motion for a collection of interacting particles. It partitions the simulation domain into small sub-domains to solve a problem [37].
- **Nekbone**: Solves 3D Poisson problem in rectangular geometry. The key MPI operations are matrix-matrix multiplication, inner products, nearest neighbor communicaton, MPI_Allreduce [15].
- **MILC**: Performs four dimensional SU(3) lattice gauge theory, mainly through near-neighbour communication and MPI_Allreduce [16].

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 [26]. Each layer serves a critical role in facilitating the efficient operation and management of the network infrastructure.

- **Data Plane :** The data plane, also known as the forwarding plane, is responsible for the actual transmission of data packets within the network [38]. 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[38]. 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,

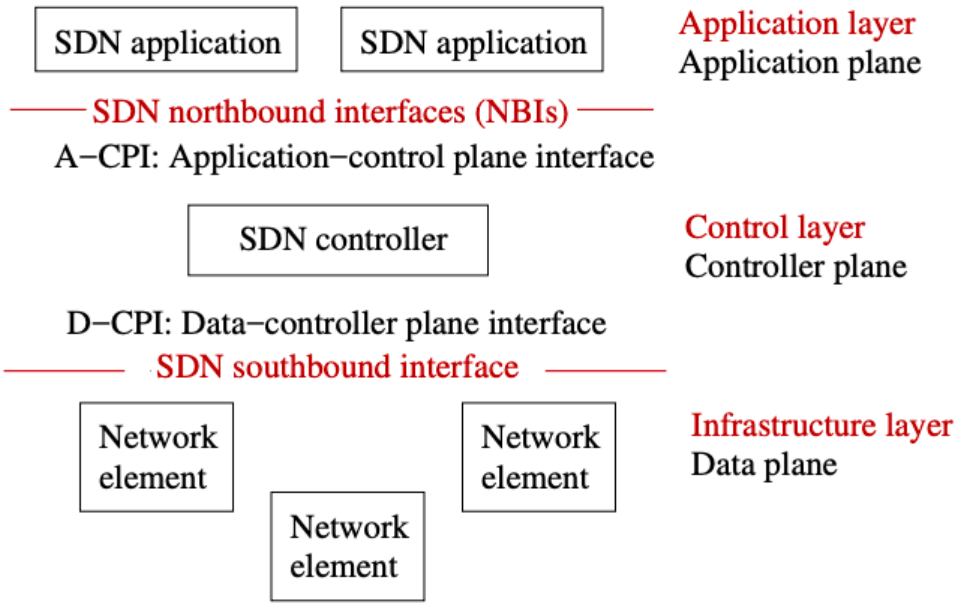

Figure 2.3: SDN abstraction

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 [38]. 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 [38, 26].
- **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 [38, 26].
- **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 [38, 26].

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 [4, 13]. One area of particular interest is the application of SDN in high-performance computing (HPC) environments. HPC systems often require fast and efficient communication between compute nodes to handle large-scale scientific computations and data-intensive workloads. By leveraging SDN, researchers aim to optimize routing and topologies in HPC environments, improving communication efficiency and resource utilization.

The structure of an SDN based HPC System (SHS) is depicted in Figure 2.4. 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 [7]. 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

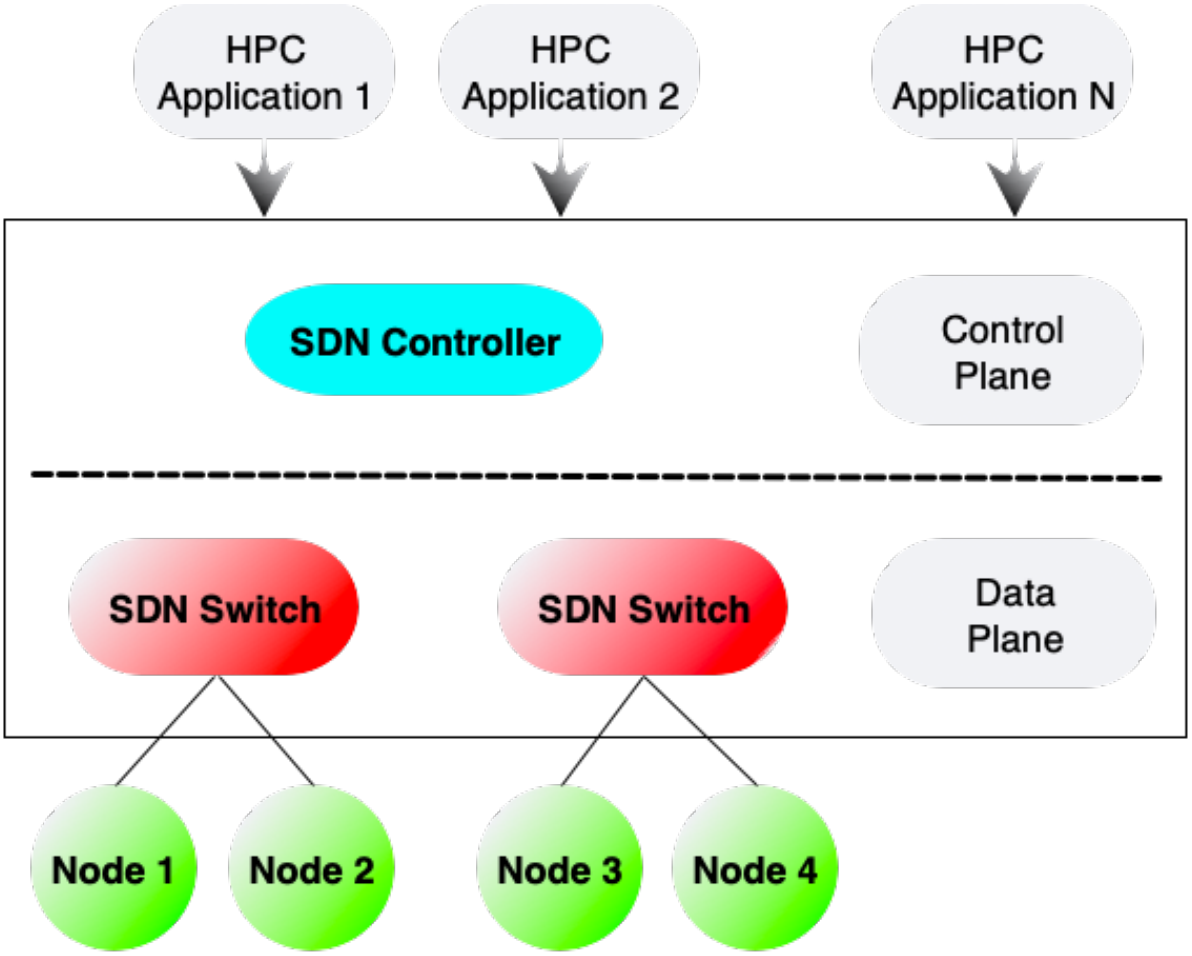

Figure 2.4: SDN based HPC System (SHS)

the SDN switches, the applications use the services provided by the SDN controller to perform communications

The remainder of this prospectus will 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 igh-performance computing (HPC) environments in the era of Software-Defined Networking.

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 hardware parameters that I pursue. To provide context for our investigation, it is essential to understand the 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, as evidenced by the comparison between Sequoia at LLNL and Summit at ORNL. This transformation necessitates a reconsideration of hardware architectural parameters to ensure that future HPC platforms can achieve optimal performance levels. I aim to address the imbalance between computation and communication capacities that arises as I 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, I utilize the TraceR-CODES simulation tool to analyze the impact of various hardware design parameters on the performance of realistic HPC workloads. Our 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 our study underscore the nuanced relationship between hardware parameters and system performance. We find 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. These findings highlight the importance of a holistic approach to designing future GPU-based HPC platforms, one that carefully considers the interplay between hardware parameters to achieve high performance. In summary, our work contributes to the ongoing dialogue on 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, I aim 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 [20]. 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 [20, 2]. 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 my new additions to TraceR-CODES, I validate 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 [29] 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). More information about the fmy applications is given in Section V.A.

We ran the three workloads in a dedicated access time (DAT) on Quartz at LLNL, during which period no other jobs ran on the machine. We 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, I use the exact system settings as Quartz: (1) I create the exact fat-tree topology as Quartz using the arbitrary graph model; (2) I 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 [31], a 5D-torus based Blue Gene/Q system. Since, the computational capabilities of Vulcan are different from Quartz, I measure the relative compute scaling factor betIen Vulcan and Quartz, and scale the computation regions of simulations accordingly. This set up helps us 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.

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 my development of the *arbitrary graph model*.

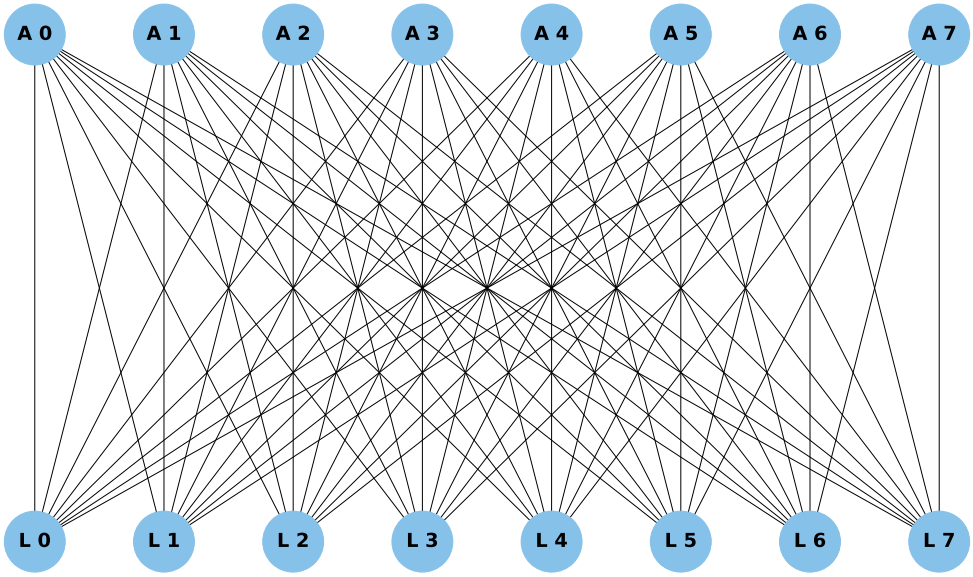

Figure 3.1: A Quartz pod with eight aggregate and eight leaf switches, and all links.

Figure 3.2 shows the results of the validation. On the horizontal axis, I 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, I compute their average error percentage to be -7.88% and present it here. We can see 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.

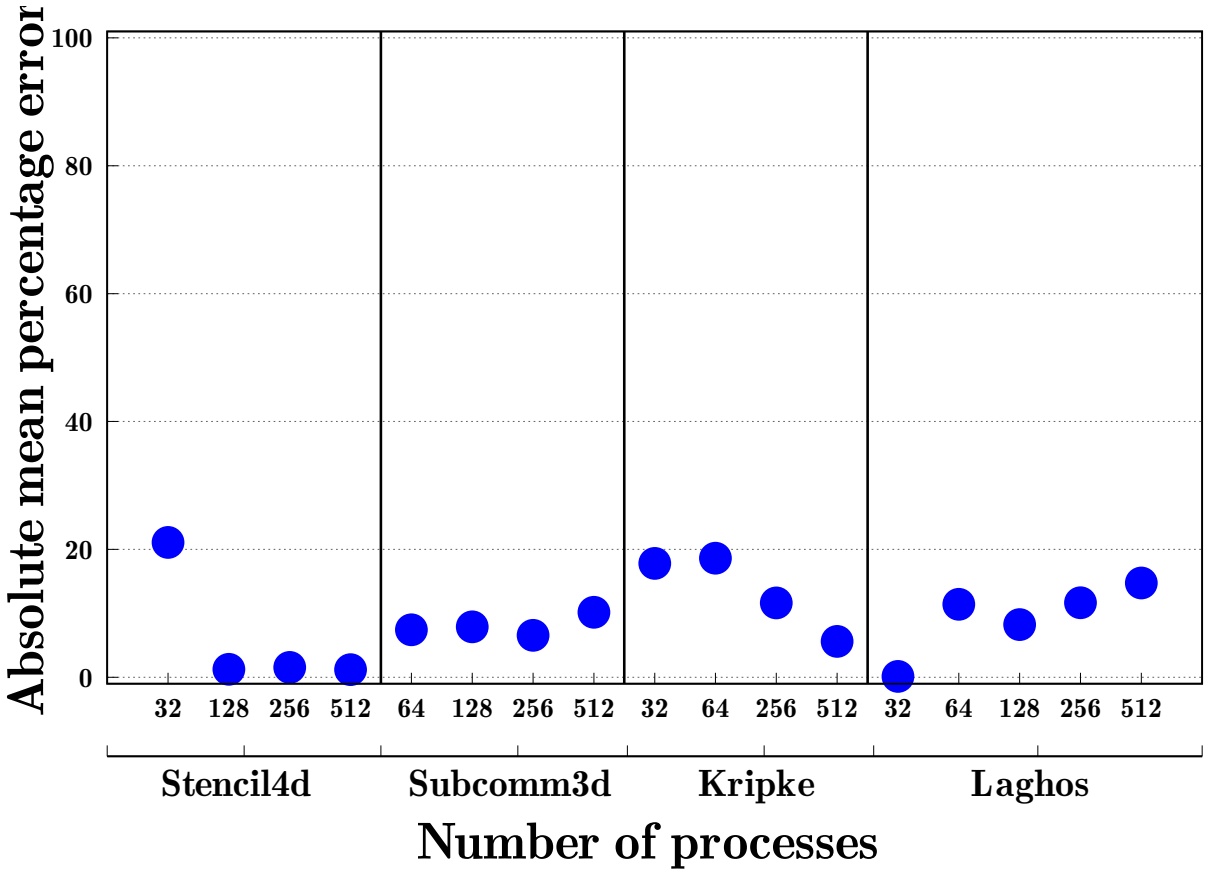

Figure 3.2: Validation of TraceR-CODES (mean percentage error in predicted runtime compared to the actual runtime).

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

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

message in-flight concurrently.

3.3 Hardware Design Parameters

Network topology: In our 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 [23] 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 [23]: 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 our experiments, I set $p = h = 8$ and $a = 16$. Each group has 16 switches and 128 compute nodes. The global link connectivity between group follows the per-router arrangement [5]. The routing algorithm used is the progressive adaptive routing (PAR) [23, 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 connects 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 our study, I vary the number of GPUs in each compute node 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 process; 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 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.

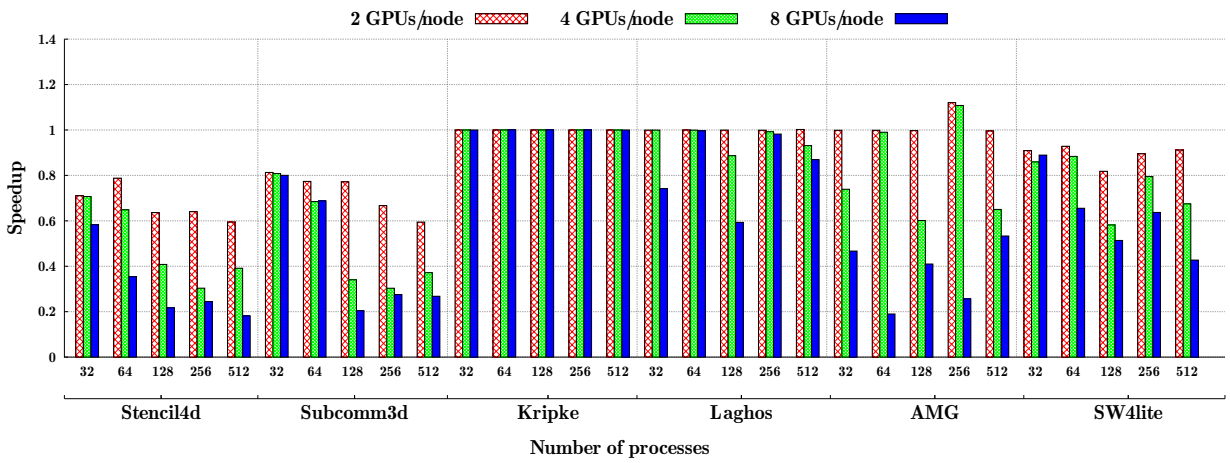

Figure 3.3: Speedup on fat-tree for various numbers of GPUs per node settings with respect to 1 GPU/node configuration.

In the experiments, when I increase the number of GPUs per node, I proportionally reduce the number of network endpoints, i.e. I make sure that for all network configurations, the total GPU count, as well the total MPI processes, is 2048. This is done to ensure that I compare systems that are of computationally equal capability as is often the case in the real world. Secondly, I make sure that each workload covers the entire network and no node is left empty during the simulation. Table 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 performance. In particular, scheduling schemes that alleviate head-of-line blocking may have significant benefits, especially when the link bandwidth is very high. In addition to head-of-line blocking, which is often mitigated by the use of virtual channels, message scheduling also affects congestion management and network utilization. Scheduling schemes that expose packets from multiple communicating-pairs to the network may perform better as it provides the network with the flexibility to use multiple network paths concurrently. To investigate the effect of message scheduling on a system with different network and different node compute capability, I compare the performance of FCFS, RR, and RR-N with different values of N on systems with different configurations.

3.4 Application and workloads

We selected six applications of different computation and communication characteristics to create realistic HPC workloads. The applications include two communication-heavy kernels, Stencil4d[9] and Subcomm3d[9], two compute-intensive applications, Kripke[28] and Laghos[30], and two applications with a balanced communication-to-computation ratio, AMG[35] and SW4lite[36] (see Figure 3.4, left). The traces used in the study were collected using Score-P [25] 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 addtion, 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 (left) 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. We tuned the communication-computation ratios 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 nding more than 95% of 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 (right)

shows how the computation-to-communication ratios change as I apply these scaling factors. 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 our study are created using the six HPC applications mentioned above at different process counts – 32, 64, 128, 256, and 512. In our 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 [19]. Our experiments use 20 such random workloads. To ensure that the reported performance of each job size of each application is representative, I ensure that each job size of each application appears at least four times in the 20 workloads. This ensures that each job size of each application has been executed under different conditions in the experiments.

3.5 Performance Study

We now present the results of simulation studies that vary different architectural parameters presented in Section 3.3.

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. We study the impact of this parameter on different types of HPC applications.

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, I see 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), I observe a noticeably different impact of changing the balance of communication to computation capability. 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, I observe a slowdown primarily with 8 GPUs per node. 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), I observe a gradual slow down when more GPUs are incorporated per node, on both network topologies. 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. We also find 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 our 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, I saw that as the number of GPUs per node increases, the default 1x network bandwidth becomes a performance bottleneck for many cases. Thus, I next study the impact of varying network bandwidth along with number of GPUs/nodes on application performance.

In our simulation experiments, I observed that the impact of network bandwidth on jobs of different sizes shares similar trends. Hence, I present data only for a job size of 128 processes. 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 requirement is 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 I need double bandwidth to get more than 90% speedup. We also see 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 GPUs per node, a significant fraction of the communication happens within the same node. This indicates that future GPU-based platforms must consider its workloads to decide important networking hardware parameters. The results for 1D dragonfly, which has 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 our 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, I only discuss results for the 1D dragonfly 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 our 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 that I use in our study 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. We conclude that 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

In this chapter, I explored the intricate dynamics of optimizing hardware parameters for future GPU-based High Performance Computing (HPC) platforms. I 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 my 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, I introduce 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.

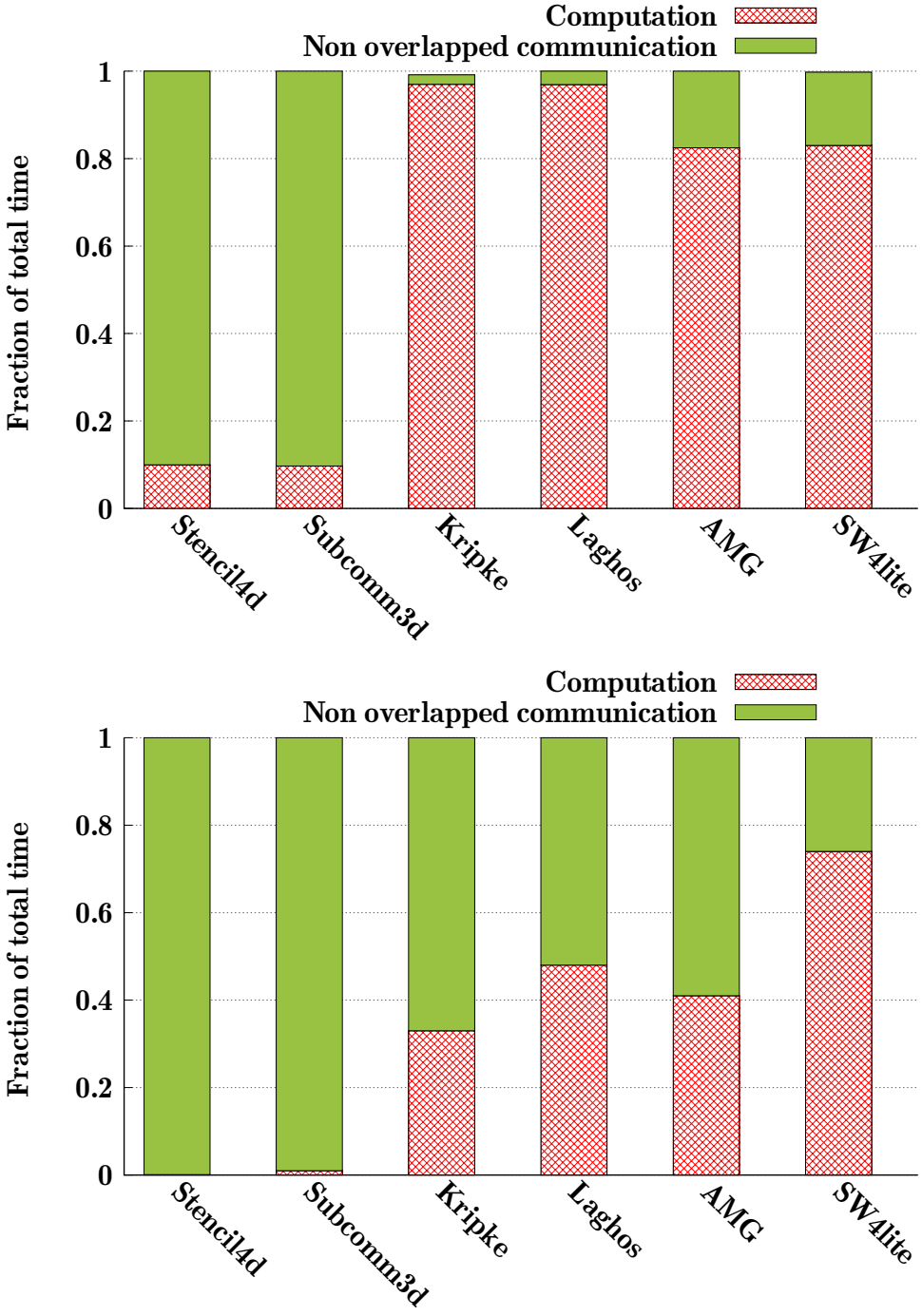

Figure 3.4: Computation and communication characteristics of all applications without scaling (left) and with scaling for GPUs (right) running on 32 processes.

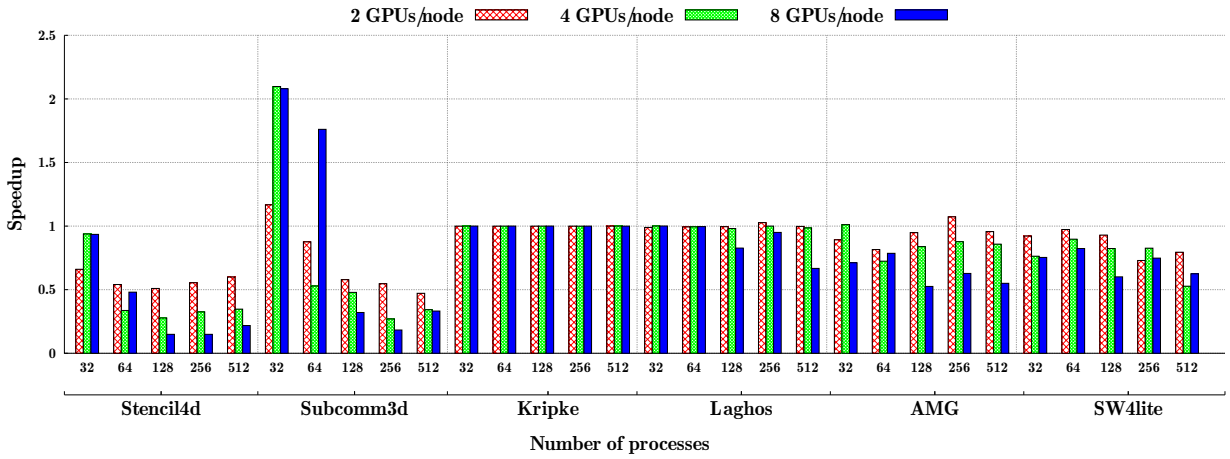

Figure 3.5: Speedup on 1D dragonfly for various numbers of GPUs per node settings with respect to 1 GPU/node configuration.

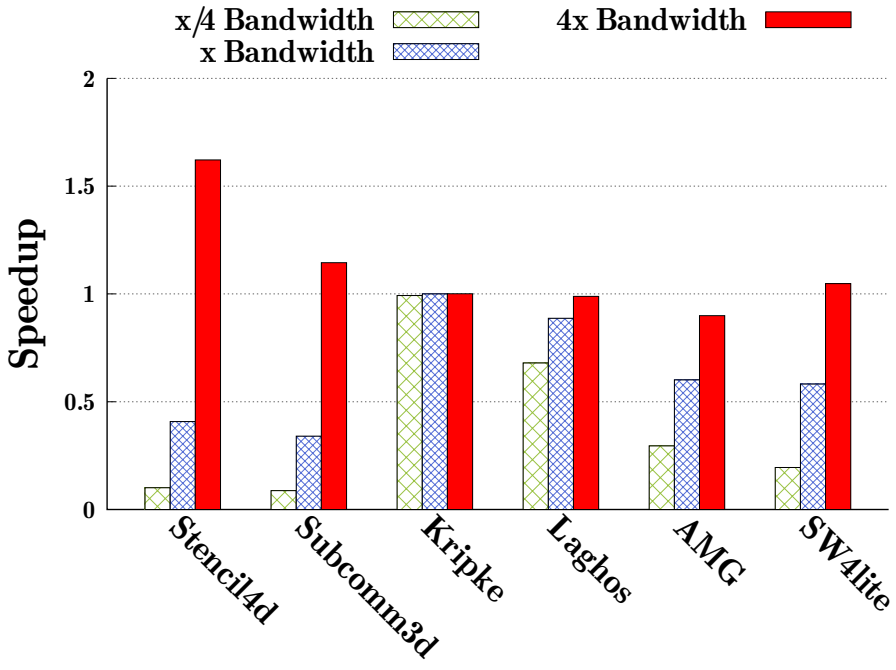

Figure 3.6: Speedup for the 4 GPUs/node configuration over 1 GPU/node in fat-tree, 1x network bandwidth configuration. Data is shown only for job sizes of 128 GPUs.

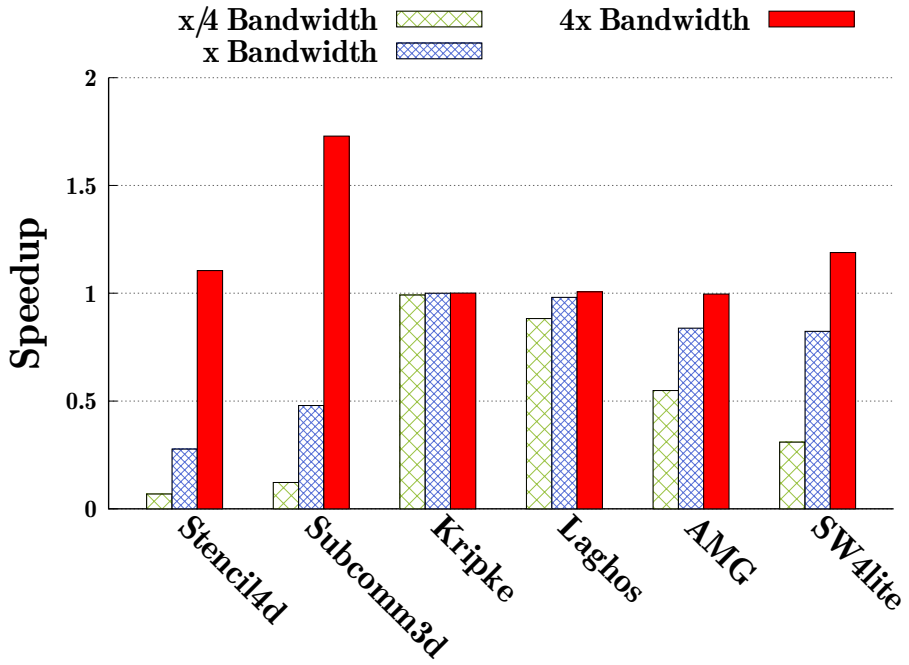

Figure 3.7: Speedup for the 4 GPUs/node configuration over 1 GPU/node in 1D dragonfly, 1x network bandwidth configuration. Data is shown only for job sizes of 128 GPUs.

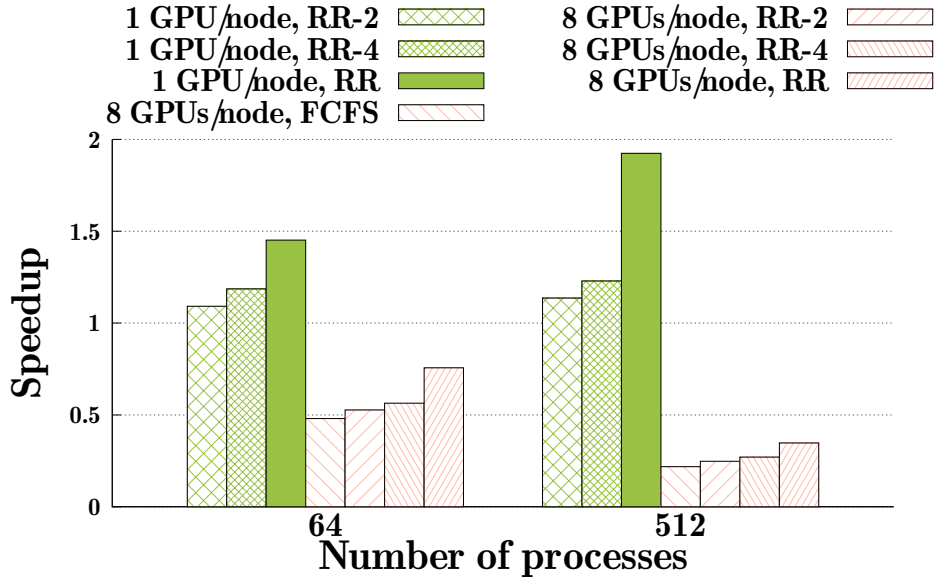

Figure 3.8: Results for Stencil4d (64 processes and 512 processes on 1D dragonfly)

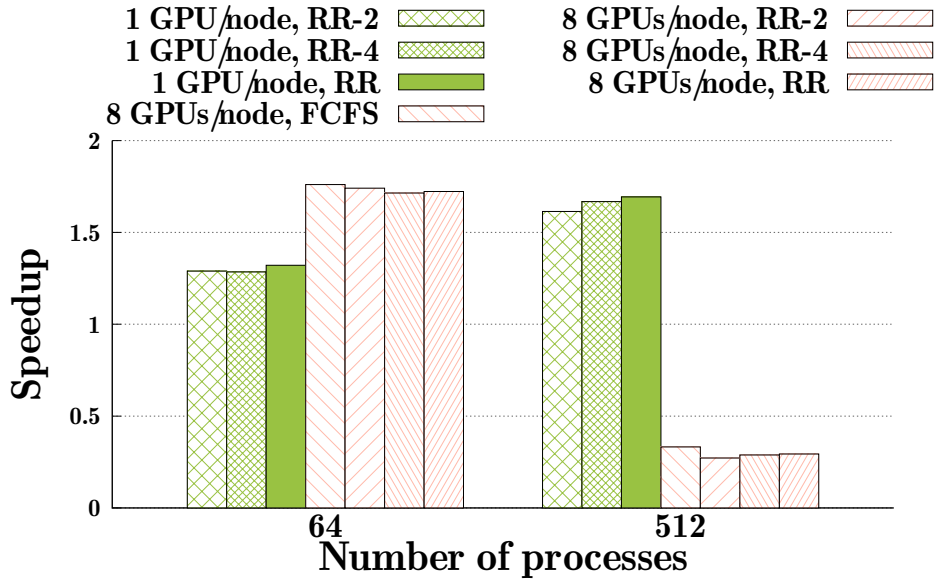

Figure 3.9: Results for Subcomm3d (64 processes and 512 processes on 1D dragonfly)

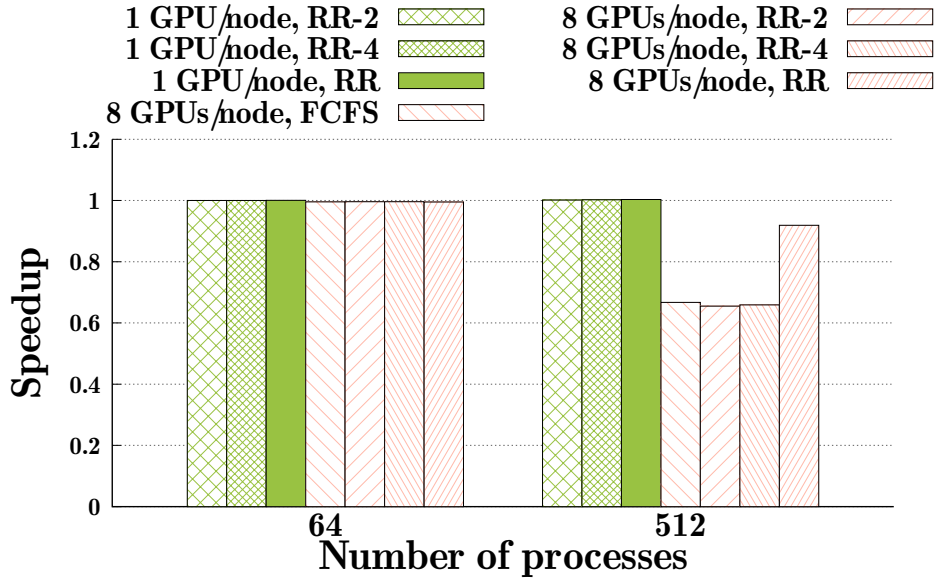

Figure 3.10: Results for Laghos (64 processes and 512 processes on 1D dragonfly)

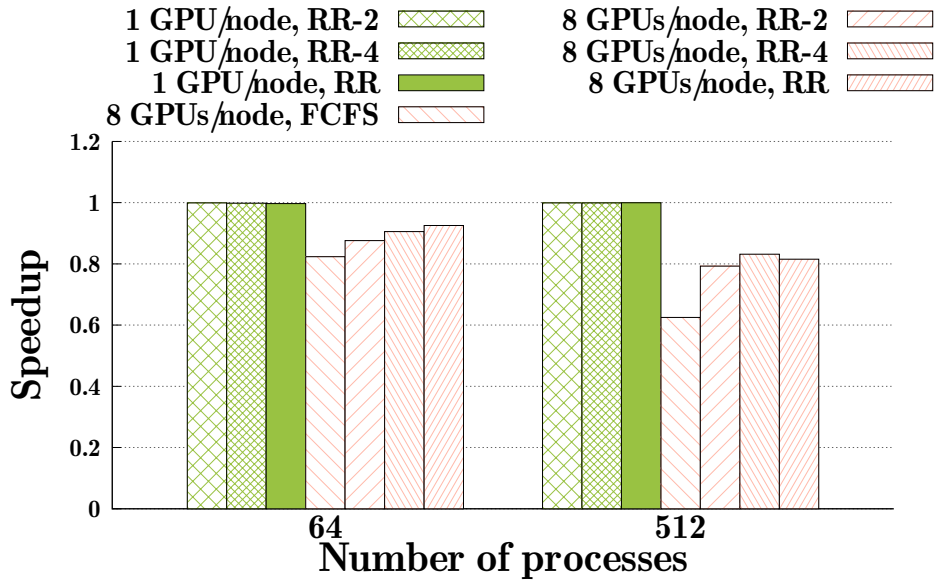

Figure 3.11: Results for SW4lite (64 processes and 512 processes on 1D dragonfly)

CHAPTER 4

TECHNIQUES TO UTILIZE SDN FOR OPTIMALLY RUNNING HPC APPLICATIONS

In this study , we tackle the challenge of incorporating Software Defined Networking (SDN) for optimally running High-Performance Computing (HPC) applications. As more data centers start to offer HPC support, there's a growing need to manage the HPC applications communication across the underlying network efficiently [26, 4, 18]. Most of these data centers use a network layout called fat-tree topology which usually has support for SDN. SDN helps by making the network smarter, so it can handle the intense network resource demands of HPC applications better. The general workload characteristics of an HPC application are vastly different from traditional data center applications. First, HPC applications are often massively parallelized, with various programming paradigms and libraries such as Message Passing Interface (MPI) [14]. The huge amuunt of computation that is required in the application is divided into smaller computations and runs on multiple computing nodes. Second, many HPC applications are tightly coupled – there are dependencies among the parallel processes and the processes need to communicate with each other during execution. Since many HPC applications simulate physical phenomena in many iterations, the applications exhibit phased behavior during execution with alternating communication and computation phases. Communications in different iterations are often the same or have similar characteristics. One common communication pattern for HPC applications is nearest-neighbor where all the ranks which are spatially near each other take part in communication.

Our main objective is to harness the power of SDN to run HPC application which are very different from data center application in Data Centers.

SDN usually has two kinds of flows elephant flows that carry large amounts of data from a source to a destination. These flows constitute a bulk of the network traffic and are sensitive to network bandwidth. Mice flows on the other hand carry small amount of data from source to destination [39, 3], and are latency sensitive.

In our work, we try to accomplish on the below objectives:

- We propose to have an network API to allow users to indicate whether a communication is an elephant flow or not. This would release the network system from classifying such flows.

- We want to create a three routing algorithm which we can implement inside our SDN controller, these routing functions help to load balance the network traffic by efficiently scheduling the bandwidth heavy elephant flows which are classfied by our API.
- Finally, we want to do a thorough simulation based evaluation of our techinques developed with static and adaptive routings which are used in fat-tree topology.

For HPC applications like the ones in Stencil4d shown in Figure 4.1, SDN users (HPC application developers, compiler, or communication library) have the knowledge whether a communication is an elephant flow or not. The code indicates that each node sends eight communication messages to its neighbors. Additionally, since the neighbors are determined only during the compile time and the set of neighbours do not change during the runtime, the user has the ability to correctly predict the communication characteristic. If the user knows the mapping of the ranks to nodes in the HPC system during the running of the application and also the data sent in the MPI calls, they can determine the communication characteristics of the application in the actual system, and can determine which flows are elephants and which are mice when a HPC application starts running on a system. Even for some communications where the destination is unknown such as the Lagos shown in Figure 4.2. The API can be used to indicate that the unknown destination flow is an elephant flow as long as the user can determine that the message size of the communication is sufficiently large. To identify an elephant flow, the information needed is the message size of the flow. Therefore, we run an application initially with a certain node to rank mapping, and collect the amount of data sent in each flow. We store this data in a matrix and then use a program to filter flows which send more than a certain threshold of data across the entire simulation. These flows are classified as elephant flows. Our API, gurantees that the user has ground truth about the communication which happens in a network when a specific set of HPC applications are run on that system.

We propose to develop a routing algorithm which aims to load balance the network traffic using the global view of a network through the SDN controller.

We intially use a greedy approach which calculates the maximum link load for all available paths for each elephant flow, and then adds whichever path has the minimum of the maximum link load -that is adds the path which is least heavily loaded for routing the elephant flow. The subnet manager provided by the SDN accumulates information about the network state and flows over the polling interval. If the SDN scheduler has no prior knowledge about the network flows, it uses D-mod-k routing. At each polling interval, the traffic classification provided by the user is forwarded

Figure 4.1: Stencil4d Code snippet

```
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 4.2: Laghos Code snippet

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

to the SDN scheduler to obtain a load-balanced routing table for the subsequent interval. This load balance schedule is a single path schedule. We also propose to develop a multipath routing, where we analyse the buffer load for k least loaded path, and use the one which has the lowest buffer load among the k paths to route our network flow. Finally, we try to develop an optimal single path routing for our research, which tries to find a non-blocking path for every elephant flow in a permutation in a full bisection bandwidth fat-tree.

We evaluate our three SDN enhanced routing schemes, two single path routing and one multi-path routing, with both the widely popular static routing scheme in fat-tree called D-mod-k and the adaptive routing which is used in fat-trees. All our routing is schemes are evaluted on two different size topologies, one mid-sized topology of 512 compute nodes and large-sized topology of 1024 compute nodes. We evalute the developed techniques across 5 different applications, which are random permutation, near neighbour, nekbone, lammps and milc; we discussed the applications in chapter 2. We chose this set of traffic patterns, as it gives a good mix of synthetic traffic, proxy application and real application, and covers all of the major HPC applications.

REFERENCES

[1] T. 500. *Top 500 list*. https://www.top500.org/. 2022.

[2] 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.

[3] 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.

[4] 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.

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

[7] 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.

[8] Keren Bergman. "Empowering Flexible and Scalable High Performance Architectures with Embedded Photonics." In: *IPDPS*. 2018, p. 378.

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

[10] 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.

[11] Daniele De Sensi et al. "Noise in the clouds: Influence of network performance variability on application scalability." In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6.3 (2022), pp. 1–27.

[12] 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.

[13] 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.

[14] Message P Forum. *MPI: A message-passing interface standard*. 1994.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

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

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

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

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

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

[32] George Michelogiannakis et al. "Bandwidth steering in HPC using silicon nanophotonics." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–25.

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

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

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

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

[37] 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.

[38] ONF Tr. "SDN architecture." In: (2016).

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