TECHNISCHE UNIVERSITÄT DRESDEN
FAKULTÄT INFORMATIK

INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR

Routing on the Channel Dependency Graph: A New Approach to Deadlock-Free, Destination-Based, High-Performance Routing for Lossless Interconnection Networks

Dissertation
zur Erlangung des akademischen Grades

Doktor rerum naturalium
(Dr. rer. nat.)

vorgelegt von

Name:	Domke	Vorname:	Jens
geboren am:	12.09.1984	in:	Bad Muskau

Tag der Einreichung: 30.03.2017
Tag der Verteidigung: 16.06.2017

Gutachter: Prof. Dr. rer. nat. Wolfgang E. Nagel, TU Dresden, Germany
Professor Tor Skeie, PhD, University of Oslo, Norway

Multicast loops are bad since the same multicast packet will go around and around, inevitably creating a black hole that will destroy the Earth in a fiery conflagration.

— **OpenSM Source Code**

Abstract

In the pursuit for ever-increasing compute power, and with Moore's law slowly coming to an end, high-performance computing started to scale-out to larger systems. Alongside the increasing system size, the interconnection network is growing to accommodate and connect tens of thousands of compute nodes. These networks have a large influence on total cost, application performance, energy consumption, and overall system efficiency of the supercomputer. Unfortunately, state-of-the-art routing algorithms, which define the packet paths through the network, do not utilize this important resource efficiently. Topology-aware routing algorithms become increasingly inapplicable, due to irregular topologies, which either are irregular by design, or most often a result of hardware failures. Exchanging faulty network components potentially requires whole system downtime further increasing the cost of the failure. This management approach becomes more and more impractical due to the scale of today's networks and the accompanying steady decrease of the mean time between failures. Alternative methods of operating and maintaining these high-performance interconnects, both in terms of hardware- and software-management, are necessary to mitigate negative effects experienced by scientific applications executed on the supercomputer. However, existing topology-agnostic routing algorithms either suffer from poor load balancing or are not bounded in the number of virtual channels needed to resolve deadlocks in the routing tables.

Using the fail-in-place strategy, a well-established method for storage systems to repair only critical component failures, is a feasible solution for current and future HPC interconnects as well as other large-scale installations such as data center networks. Although, an appropriate combination of topology and routing algorithm is required to minimize the throughput degradation for the entire system. This thesis contributes a network simulation toolchain to facilitate the process of finding a suitable combination, either during system design or while it is in operation. On top of this foundation, a key contribution is a novel scheduling-aware routing, which reduces fault-induced throughput degradation while improving overall network utilization. The scheduling-aware routing performs frequent property preserving routing updates to optimize the path balancing for simultaneously running batch jobs. The increased deployment of lossless interconnection networks, in conjunction with fail-in-place modes of operation and topology-agnostic, scheduling-aware routing algorithms, necessitates new solutions to solve the routing-deadlock problem. Therefore, this thesis further advances the state-of-the-art by introducing a novel concept of routing on the channel dependency graph, which allows the design of an universally applicable destination-based routing capable of optimizing the path balancing without exceeding a given number of virtual channels, which are a common hardware limitation. This disruptive innovation enables implicit deadlock-avoidance during path calculation, instead of solving both problems separately as all previous solutions.

Contents

List of Figures **xi**

List of Tables **xiii**

List of Algorithms **xv**

List of Abbreviations and Symbols **xvii**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Contributions 4
- 1.3 Thesis Organization 5

2 Related Work **7**
- 2.1 State-of-the-Art for High-Performance Interconnect Designs 7
  - 2.1.1 Resiliency of Networks, Topologies, and Routings 7
  - 2.1.2 HPC Routing Strategies and Network Management 8
  - 2.1.3 Simulation Frameworks for Network Analyses 9
- 2.2 State-of-the-Art for Deadlock-free Routing Approaches 10
  - 2.2.1 Topology-aware Routing Algorithms 12
  - 2.2.2 Topology-agnostic Routing Algorithms 12

3 Background, Assumptions, and Definitions **15**
- 3.1 Interconnection Networks and Routing Algorithms 15
  - 3.1.1 Network-related Definitions 15
  - 3.1.2 Routing-related Definitions 17
  - 3.1.3 Topology and Routing Metrics 19
- 3.2 Common Network Topologies and Production HPC Systems 21
  - 3.2.1 Meshes and Tori 22
  - 3.2.2 Fat-Trees and eXtended Generalized Fat-Trees 23
  - 3.2.3 Kautz Graph 24
  - 3.2.4 Dragonfly Topologies 24
  - 3.2.5 Random Topologies 26

3.2.6 Real-world HPC Systems . . . 26
3.3 Selection of Routing Algorithms for HPC . . . 28
3.3.1 MinHop Routing . . . 28
3.3.2 Single-Source Shortest-Path Routing (SSSP) . . . 28
3.3.3 Dimension Order Routing (DOR) . . . 29
3.3.4 Up*/Down* and Down*/Up* Routing . . . 29
3.3.5 Fat-Tree Routing . . . 29
3.3.6 Layered Shortest Path Routing (LASH) . . . 31
3.3.7 Torus-2QoS Routing . . . 31
3.3.8 Deadlock-Free SSSP Routing (DFSSSP) . . . 32

4 Fail-in-Place High-Performance Networks 33
4.1 Failures Analysis for Real HPC Systems . . . 33
4.2 Building Blocks for Fail-in-Place Networks . . . 36
4.2.1 Resilient Topologies . . . 36
4.2.2 Resilient Routing Algorithms . . . 36
4.2.3 Resiliency Metrics . . . 37
4.3 Interconnect Simulation Framework . . . 40
4.3.1 Toolchain Overview . . . 40
4.3.2 InfiniBand Model in OMNeT++ . . . 41
4.3.3 Traffic Injection . . . 42
4.3.4 Simulator Improvements . . . 43
4.4 Simulation Results . . . 44
4.4.1 Initial Usability Study . . . 44
4.4.2 Influence of Link Faults on Small Topologies . . . 46
4.4.3 Influence of Link Faults on Large Topologies . . . 50
4.4.4 Case Study of the TSUBAME2 Supercomuter . . . 52
4.4.5 Case Study of the Deimos HPC System . . . 54

5 Utilization Improvement through SAR 57
5.1 Conceptual Design of SAR for shared HPC Environments . . . 57
5.2 Example Implementation of the Scheduling-Aware Routing . . . 62
5.2.1 Hardware and Software Building Blocks . . . 62
5.2.2 Scheduling-Aware Routing Optimization . . . 65
5.2.3 Property Preserving Network Update for InfiniBand . . . 67
5.3 Evaluation . . . 71
5.3.1 Current Limitations and Problems . . . 72
5.3.2 Theoretical Evaluation of Network Metrics . . . 72
5.3.3 Practical Evaluation on a Production System . . . 77

6 Routing on the Channel Dependency Graph 81
6.1 Limitations of Multi-Step Routing Approaches 81
6.2 Nue Routing 84
6.2.1 Complete Channel Dependency Graph 84
6.2.2 Escape Paths for the Static Nue Routing 85
6.2.3 Choosing Root Node for the Spanning Tree 87
6.2.4 Dijkstra’s Algorithm for the complete CDG 89
6.2.5 Nue Routing Function 91
6.2.6 Optimizations for Nue Routing 92
6.2.7 Correctness, Completeness & Complexity 99
6.3 Evaluation of Nue Routing 102
6.3.1 Path Length and Edge Forwarding Index 103
6.3.2 Throughput for Regular and Irregular Topologies 104
6.3.3 Runtime and Practical Considerations 107

7 Summary, Conclusion, and Future Work 109

Bibliography 113

List of Figures

1.1 Interconnect technology statistics of the TOP500 list of November 2016 2

3.1 5-node interconnection network $I = G(N,C)$ 16
3.2 Using a shortest-path, counter-clockwise routing for network I induces the channel dependency graph D 18
3.3 Example visualization of a 3-dimensional mesh and torus topology without terminals . . 22
3.4 Example visualization of a k-ary n-tree topology and an extended generalized fat-tree topology 23
3.5 Example visualization of a Kautz graph and a dragonfly topology 24
3.6 Topology configurations of three real-world supercomputers 27

4.1 Network-related hardware failures of three HPC systems 34
4.2 Whisker plot for lost connections while removing links or switches from a 10-ary 3-tree without performing a rerouting 39
4.3 Toolchain to evaluate the network throughput of a fail-in-place network 41
4.4 Whisker plots of consumption bandwidth for uniform random injection 47
4.5 Histograms for exchange patterns with error bars, showing mean value and the 95% confidence interval for ten seeds 49
4.6 Histograms for exchange patterns executed on networks with up to 8% link failures . . . 51
4.7 Histograms for exchange patterns for different failure types using DFSSSP, fat-tree and Up*/Down* routing on TSUBAME2 53
4.8 Histograms for exchange patterns for different failure types using MinHop, DFSSSP and Up*/Down* routing on Deimos 55

5.1 Batch job history of two petascale HPC systems for one month of operation 58
5.2 Comparison of effective EFI for inter-switch links: Oblivious routing vs. scheduling-aware routing for three equal-sized batch jobs on a 2-level fat-tree 60
5.3 Qualitative comparison of different routing and network optimizations for supercomputers 61
5.4 SLURM architecture 63
5.5 Open MPI's modular component architecture establishes a message passing interface between application and hardware to enable inter-process communication 64
5.6 Flowchart of a filtering tool 65

5.7 Sequence diagram of a five-phase update protocol to achieve property preserving network updates of InfiniBand networks 70
5.8 Replay of job history (Figure 5.1) for two HPC systems 74
5.9 Heat map of the EFIs for inter-switch links of TSUBAME2 for the 200-node batch job on day 16 of Feb. ’15; Used routing algorithm: fat-tree 76
5.10 Heat map of the effective EFIs for inter-switch links of TSUBAME2 for the 200-node batch job on day 16 of Feb. ’15; Used routing algorithm: SAR 76
5.11 Histograms of effective EFI for inter-switch links: Fat-tree routing vs. scheduling-aware routing for the 200-node batch job on TSUBAME2 77
5.12 Runtime measurement for MPI_Alltoall (with 1 MiByte in each send buffer) on 28 compute nodes using three different routing approaches 79
5.13 Visualization of the network update protocol (without QP draining) and path migration between two inter-switch links on a testbed during high MPI load 80

6.1 Using a shortest-path routing for 5-node ring network I induces the CDG D 82
6.2 Simulated throughput for an all-to-all operation and required VCs for deadlock-freedom for different routing algorithms 83
6.3 Complete CDG $\overline{D}$ for the 5-ring network with shortcut; All channels are in unused state . 84
6.4 Sketching the process of finding paths between network nodes within the complete CDG $\overline{D}$ while avoiding to close any cycles in the induced CDG D 85
6.5 Acyclic escape paths D^{τ} for the 5-node ring network with shortcut 87
6.6 Arrows show the initial channel dependencies of the escape paths for the node set $N_i^d = \{n_1, n_2, n_3\}$ and root node n_5 or root node n_2, respectively 88
6.7 State change of $\overline{D}$ after five steps with Algorithm 6.2, starting from c_{n_1,n_2} 93
6.8 Impasse of Algorithm 6.2 to reach n_4 based on previously placed routing restrictions for channels dependencies $(c_{n_1,n_3}, c_{n_3,n_4})$ and $(c_{n_7,n_5}, c_{n_5,n_4})$ 95
6.9 States of the channel dependency graph $\overline{D}$, of the subnetwork I^* of Figure 6.8a, shown . 98
6.10 Averaged edge forwarding index metrics for 1,000 random topologies with 125 switches, 1,000 terminals, and 1,000 switch-to-switch channels 104
6.11 Simulated throughput for an all-to-all operation on five standard and three real-world topologies, as configured according to Table 6.1 106
6.12 Runtime comparison of deadlock-free routings for 3-dimensional tori (without channel redundancy) of various topology sizes with 1% injected link failures 108

List of Tables

2.1 Qualitative comparison of existing topology-aware and topology-agnostic routing algorithms for supercomputers 14

3.1 Vertex-connectivity for the investigated network topologies of Sections 3.2.1 – 3.2.4 (except for random and Cascade) 25

4.1 Comparison of network-related hardware and software failures, MTBF/MTTR, and annual failure rates for three production supercomputers 34
4.2 Topology configurations for a balanced (and unbalanced) number of HCAs with link redundancy $r_{C'}$ to allow a fair comparison between topology types 45
4.3 Usability of topology and routing combinations for networks listed in Table 4.2 45
4.4 Intercept [in Gbyte/s], slope, and R^2 for balanced topologies of Table 4.2 for best performing routing algorithm 50
4.5 Intercept [in Gbyte/s], slope, and R^2 for the large topology/routing combination shown in Figure 4.6c 52
4.6 Intercept, slope, and R^2 for TSUBAME2 and Deimos shown in Figure 4.7c and Figure 4.8c 56

5.1 Improvements by the scheduling-aware routing compared to DFSSSP, fat-tree, and Up*/Down* routing for the Taurus HPC system 75
5.2 Improvements by the scheduling-aware routing compared to DFSSSP, fat-tree, and Up*/Down* routing for the TSUBAME2 HPC system 75

6.1 Topology configurations, with link redundancy $r_{C'}$, used for throughput simulation results visualized in Figure 6.11 105

List of Algorithms

3.1 SSSP: Global balancing of routes with single-source shortest-path routing 28
3.2 Fat-Tree: Setting LFTs towards all compute nodes 30
3.3 Fat-Tree: Assign-down-going-port-by-ascending 30
3.4 Fat-Tree: Assign-up-going-port-by-descending 30
3.5 LASH: Deadlock-free path calculation by layered shortest path routing 31
3.6 DFSSSP: Search cycles in the CDGs and remove potential deadlocks 32

5.1 Scheduling-aware DFSSSP routing 67

6.1 Convex subgraph for node set N^d of an unweighted graph $G(N,C)$ 89
6.2 Modified Dijkstra's algorithm within the complete CDG $\overline{D}$ 90
6.3 Nue routing calculates all paths within a network $I = G(N,C)$ for a given number of virtual channels $k \geq 1$ 91
6.4 Search for cyclic *used* subgraphs in the complete CDG $\overline{D}$ 94
6.5 Local backtracking finds paths into islands 96
6.6 Finding shortcuts through island nodes 98

List of Abbreviations and Symbols

ACK	Acknowledgement signal to acknowledge the receipt of data
API	Application programming interface
APM	Automatic path migration
BFS	Breadth-first search
BGP	Border Gateway Protocol
BTL	Byte transfer layer of Open MPI
CDG	Channel dependency graph
CEE	Converged enhanced Ethernet
DDR	Double data rate – 5 Gb/s per lane (or 20 Gb/s for 4x link width)
DFS	Depth-first search
DFSSSP	Deadlock-free single-source shortest-path routing
DOR	Dimension order routing
EDR	Enhanced data rate – 25 Gb/s per lane (or 100 Gb/s for 4x link width)
EFI	Edge forwarding index
FDR	Fourteen data rate – 14 Gb/s per lane (or 56 Gb/s for 4x link width)
HCA	Host channel adapter (InfiniBand's network interface card)
HPC	High-performance computing
IB	InfiniBand
IBTA	InfiniBand Trade Association
LASH	Layered shortest path routing
LFT	Linear forwarding tables

LID	Local identifier
LMC	LID mask control
MAD	Management datagram packet of InfiniBand
MCA	Modular component architecture
MPI	Message Passing Interface
MTBF	Mean time between failure
MTTR	Mean time to repair
MTU	Message transfer unit
MUD	Multiple Up*/Down* routing
NIC	Network interface card
NoC	Network-on-Chip
PCI	Peripheral Component Interconnect
QDR	Quad data rate – 10 Gb/s per lane (or 40 Gb/s for 4x link width)
QP	Queue pair
RC	Reliable connection
SAR	Scheduling-aware routing
SDN	Software defined network
SDR	Single data rate – 2.5 Gb/s per lane (or 10 Gb/s for 4x link width)
SLURM	Simple Linux Utility for Resource Management
SQ	Send queue
SQD	Send queue draining
SSSP	Single-source shortest-path routing
UC	Unreliable connection
VC	Virtual channel
VL	Virtual lanes (InfiniBand's virtual channel)

WR	Work request
XGFT	eXtended Generalized Fat-Tree
χ	Network state, i.e., the entirety of all forwarding tables of all switches
Δ	Maximum degree of the interconnection network, i.e., maximal switch radix
$\gamma(I,R)$	Edge forwarding index of the network I as defined by the routing R
$\gamma^e(I,R,J)$	Edge forwarding index of the network I as defined by the routing R and jobs set J
$\gamma^e_{R,J}(c)$	Effective edge forwarding index of a channel $c \in C$ w.r.t job set J
$\gamma_R(c)$	Edge forwarding index of a channel $c \in C$ defined by the routing R
κ/λ	Vertex-/edge-connectivity for a graph or network
${}^sP_{n_u,n_v}$	Shortest path between two network nodes $n_u, n_v \in N$
$\omega(\cdot)$	Function assigning a unique identifier to a cycle-free subgraph of $\overline{D}_i$
$\overline{D}_i$	Complete channel dependency graph $G(C_i,\overline{E}_i)$ of the i-th virtual layer L_i
$\overline{E}_i$	Set of edges ($\subseteq C_i \times C_i$) of the complete channel dependency graph $\overline{D}_i$
$\tau(I)$	Spanning tree of network I
$\theta_{R,J}$	Dark fiber percentage of unused channels for intra-job routes as induced by the routing R
$\tilde{c}$	Virtual channel of a network channel $c \in C$
C	Set of network channels or links
c	Network link or channel of the channel set C
C^*	Set of switch-to-switch channels
C_i	Set of network channels of the i-th virtual layer L_i
C_L	Set of faulty switch-to-switch channels
$C_B(n)$	Betweenness centrality value of a network node $n \in N$
D_i^τ	Escape paths defining a cycle-free subgraph of $\overline{D}_i$
h	Path length
H_i	Convex Subgraph $G(N_i^H, C_i^H)$ for a set $N_i^d \subseteq N$ of network nodes

h_R	Routing-induced path length or routing-based hop count between two nodes
I	Interconnection network with $I := G(N,C)$
J	Set of batch jobs running simultaneously on a supercomputer
k	Maximum number of supported virtual channels (except when used for k-ary n-trees)
N	Set of network nodes composed of switches and terminals, i.e., $N = S \cup T$
n	Network node of the node set N
N_i^d	Set of destination nodes being routed with the i-th virtual layer L_i
N_j	Set of network nodes belonging to the same batch job $j \in J$
P_{n_u,n_v}	Path between two network nodes $n_u, n_v \in N$
Q	Queue data structure used within algorithms
R	Routing function
R^*	Perfectly balanced routing function (with respect to the edge forwarding index)
R^2	Coefficient of determination for a linear regression
$r_{C'}$	Link redundancy for the augmented network $I' := G(N,C')$
S	Set of switches in the network (subset of N)
T	Set of terminal nodes (subset of N)
u_s	Sequence of atomic updates applied to forwarding tables

1 Introduction

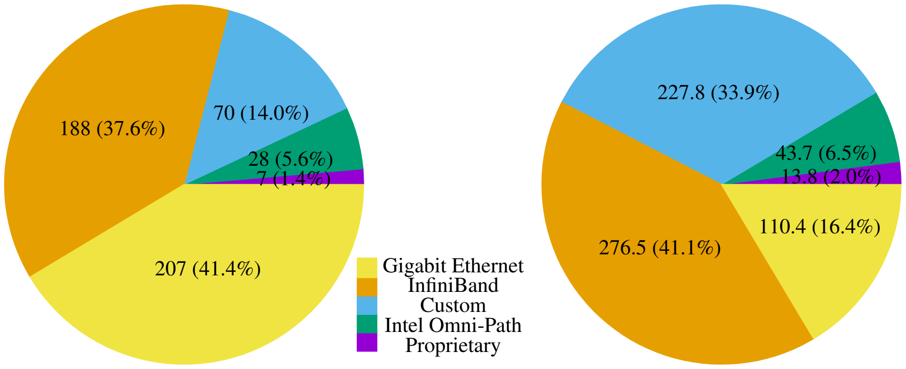
Ensuring efficient communication within a supercomputer is an important task to enable progress of other scientific fields. High-speed lossless interconnection networks, such as InfiniBand, are the de facto standard for the fastest computers in the world, see Section 1.1. This thesis advances the state-of-the-art, as listed in Section 1.2, by introducing new concepts for operating these networks, both in terms of hardware and software. Section 1.3 provides the list of preceding articles published by the author of this thesis, and outlines the high-level structure and content of the following chapters.

1.1 Motivation

With Moore's law slowly coming to an end, many information technology domains started to scale-out. A global trend visible from small many-core systems-on-chip, such as the 256-core Kalray MPPA-256 chip [Din+13] or the 1024-core Epiphany-V developed by Adapteva [Olo16], over large-scale data centers or data wearhouses [Mic15], to the world's most powerful supercomputers, such as the 40,960-node Sunway TaihuLight system hosting more than 10 million compute cores [Don16]. All these systems, and many more (e.g., wireless sensor networks, the Internet with the emerging Internet-of-Things, etc.), have one common property: the requirement for a network connecting the individual devices. The network, connecting these compute nodes, machines, or appliances, plays a crucial role for successful operation.

For high-performance computing (HPC) systems, it can be expected that the number of network endpoints will grow significantly [KBB08; Luc+14] which emphasizes the role of the interconnection network as one of the most critical components in a supercomputer even more. These HPC networks, as well as other previously mentioned networks, are largely controlled by routing algorithms which determine how to forward packets. Routing algorithms have to balance multiple, partially conflicting, requirements. For example, they shall provide the best forwarding strategy to guarantee a certain quality of service (throughput, latency, etc.), which is an NP-hard problem in general, while minimizing the runtime of the routing in order to quickly react to failures of network components.

Most supercomputers listed in the TOP500 list of November 2016 [Str+16], 84.6% to be precise, rely on standardized and off-the-shelf interconnection network components. The deployed interconnection technologies are either different high-speed versions of Ethernet [IEE16] (207 systems), different speeds of InfiniBand [Inf15] (188 systems), or Intel's 100 Gbit/s Omni-Path technology [Bir+15] (28 systems), see Figure 1.1a. Both, the system share and performance share shown in Figure 1.1 are adjusted to include the current No. 1 system, the Chinese 93 Pflop/s Sunway TaihuLight supercomputer, in the InfiniBand (IB)



(a) Interconnect family system share **(b)** Performance share [in Pflop/s]

Figure 1.1: Interconnect technology statistics of the TOP500 list of November 2016; Ethernet distribution: 1x 100G, 178x 10G, 28x 1G; InfiniBand distribution: 15x EDR, 149x FDR, 24x QDR; Shares adjusted for IB-based Sunway TaihuLight (No. 1)

shares, since it utilizes Mellanox's EDR InfiniBand switch chips and host channel adapters [Don16]. Otherwise, these numbers would unfairly belittle the significance of the InfiniBand for the performance of the top-tier supercomputers. Besides Ethernet or InfiniBand interconnects, a subset of HPC systems make use of either custom or proprietary interconnects, such as Cray's Gemini interconnect used in Titan [ARK10] (No. 3) or the Tofu interconnect of the K computer [Aji+12a] (No. 7).

To meet the vast data transfer demands of supercomputers in terms of low latency and high bandwidth, these network architectures increasingly embrace functions that enable a tighter interaction between networking hardware and programming models. The best examples are Remote Direct Memory Access (RDMA) in combination with a kernel bypass, and lossless Layer 2 networking, all tailored for these demands. The former enables the programmer to directly instruct the network interface to read and write remote memory without having to issue a kernel call to the operating system. While the latter enables the reliable transport within the network which is needed for RDMA. Such advanced features have been prevalent in the high-speed networking area, such as InfiniBand [SWM03] or Cray's Cascade [Faa+12], but also find their way into Ethernet, which was recently extended with Priority Flow Control at Layer 2 to support RDMA [Inf10; Zhu+15].

Unfortunately, as network topologies grow, failures of switches and connectors or cables become common. As opposed to network endpoints, the wiring complexity and infrastructural demands of cabling, e.g., arrangement of cable trays, make maintaining complex networks challenging and expensive. Thus, network structures are often kept in place over years and multiple generations of machines while other components such as CPU or main memory are upgraded. Today, many networks are based on the concept of over-provisioning the hardware, such as installing spare cables in the cable trays or having an inventory of spare parts. Alternatively, they are operated in a deferred repair mode, in which a failed component will not be replaced instantaneously but within a reasonable time frame, such as a business day.

Fail-in-place strategies are common in storage systems when maintenance costs exceed maintenance benefits, such as in large-scale data centers with millions of hard drives. For example, Microsoft owned more than one million servers in 2015 [Mic15], i.e., even an optimistic failure rate of 1% per year and two hypothetically hard drives per server would result in a mean time between failure of 26 minutes. Instead of replacing the hard drives of a server, the storage system, such as IBM's Flipstone [Ban+08], uses RAID arrays for reliability and a software approach to disable failed hard drives and to migrate the data, until a critical component failure disables the entire server. Hence, a fail-in-place approach for interconnection networks could be similarly beneficial, if supported by the correct combination of network topology and applied routing algorithm.

Routing algorithms for HPC systems have been the topic of many studies ranging from topology-specific routing algorithms [Rod+09; Zah12] through general deadlock-free algorithms [DS87; Sch+91], more advanced deadlock-free algorithms balancing the routes [DHN11], to advanced path-caching for quick failover [VSR15]. A good overview is provided by Flich et al. [Fli+12]. Many advanced approaches for application-specific [Pal+06; Kin+09] or topology-specific [Sub+12; Pri+13b; Jok+15] routing and mapping assume idealized conditions such as a regular topology without faulty components, isolated bulk-synchronous applications communicating in synchronized phases, and the absence of system noise. Unfortunately, these assumptions are rarely true in practice. Presumably, integrating additional information about the current state of the supercomputer, such as state of the batch system (i.e., the current job mix) or application demands, can help guiding the network manager in assigning optimized routes on-demand which potentially increases utilization and achievable throughput.

Due to the previously addressed hardware failures and these on-demand software adjustments, topology-aware routing algorithms become increasingly inapplicable in modern supercomputers. Hence, avoiding deadlock situations [CES71; DT03], a requirement for lossless Layer 2 networking, becomes much harder to achieve. A network deadlock arises when a group of packets cannot be forwarded because of unavailable resources, such as buffers or channels, and the group is circularly depended on each other, which disables parts or the entire network. Topology-aware routings usually avoid deadlocks algorithmically for many regular topologies, e.g., by restricting the routing to use only a subset of all available channel dependencies, as it is implemented by dimension-order routing (DOR) [RR10]. In contrast, many topology-agnostic routing technique for HPC and on-chip networks (NoC) use virtual channels to break deadlocks [SLT02; BM06; Shi+09; DHN11]. Yet, all these concepts have limitations:

(1) Topology-aware routing algorithms usually assume perfect topologies and often do not support switch or link failures, see Chapter 4 of this thesis,

(2) Cycle-avoiding routing algorithms often cannot balance routes across the available links and thus limit global achievable throughput [SRD00a], and

(3) Routing algorithms based on virtual channel isolation fail when the required number of virtual channels is not available [Fli+12].

1.2 Contributions

This thesis contributes to multiple frontiers of the design and operation of interconnection networks of modern and future HPC systems, as well as proposes a novel approach to solve the deadlock-free routing problem of these networks, which will greatly influence the next generation of routing algorithms.

Historical failure data of three production HPC systems allows this thesis an analysis that subsequently enables a realistic performance model for a wide range of HPC networks. Initial connectivity simulations reveal that it is insufficient to consider fault properties of the topology or to consider the resiliency of the routing algorithm in isolation. For fail-in-place networks, both these properties and the combination of topology and routing must meet requirements. The developed flit-level simulation toolchain for InfiniBand networks determines the achievable throughput accurately. An exhaustive study, using the simulator for several real-world topologies and two real HPC systems, shows that the choice of the routing algorithm influences the number of disconnected paths in case of a failure and the quality of service after a reinitialization of the network with missing links or switches. Depending on the underlying topology and routing method, a low number of failing links already reduces the overall throughput by up to 30% (e.g., for small-scale fat-trees), or an HPC system can be operated in fail-in-place mode with almost zero performance degradation, respectively. Overall, the study reveals that fail-in-place network designs can be accomplished with an appropriate combination of topology and routing algorithm, while assuming that a performance degradation up to a predefined threshold is tolerable. Changing the used routing method of the two investigated HPC systems would not only increase their network throughput by up to 2.1x or 3.1x for the fault-free network, respectively, but also increase their fail-in-place characteristics.

The influence of the degrading fail-in-place interconnect on the achievable application throughput can be further reduced by optimizing the routing algorithm to exploit the knowledge about the momentary resource allocation in the supercomputer. The developed scheduling-aware routing (SAR), an enhancement of one of the most appropriate routings for fail-in-place networks, frequently reconfigures the switches' routing tables for intra-job communication of concurrently running applications. This low-overhead routing algorithm is already deployed on a production petascale supercomputer for over one year and improves the system's utilization and its amount of scientific output. Two newly introduced metrics, the effective edge forwarding index and the dark fiber percentage, show the effectiveness of SAR in comparison to the state-of the-art oblivious routing mechanisms for HPC systems, using simulations with historical resource allocations of two InfiniBand-based supercomputers. The former metric measures the well-established edge forwarding index with respect to the running applications, both within and between allocations, to estimate a theoretical upper bound for the worst-case network congestion. The latter metric, the dark fiber percentage, quantifies the utilization of the network. Unfortunately, some network architectures, such as InfiniBand, rely on in-order packet delivery, i.e., a prerequisite for unobstructed operation, which can be compromised by frequent reconfigurations. Hence, the formalism of per-packet consistent updates is extended for in-order delivery and deadlock-freedom, and a new network update protocol is introduced to achieve the required consistency for InfiniBand hardware.

Even more important than preventing deadlocks during the reconfiguration of the network is the deadlock-avoidance while a single network configuration is active. A thought experiment visualizes the limitations of state-of-the-art multi-step routing approaches, i.e., the impossibility to achieve deadlock-freedom for shortest-path routing with limited amount of supported virtual channels on arbitrary or faulty topologies. Consequently, a novel routing approach is developed based on the concept of routing within the complete channel dependency graph that overcomes all limitations (1) – (3) mentioned in Section 1.1. Hence, this routing, called Nue, is capable of distributing the paths across the network to increase the throughput, while reliably avoiding deadlocks. Flit-level simulations with the developed toolchain show that the destination-based Nue routing works on any arbitrary topology, especially fail-in-place networks, with all possible numbers of available virtual channels, including network technologies without support for virtual channels. The conducted complexity analysis, as well as runtime measurements with an implementation of Nue in the InfiniBand subnet manager, demonstrate Nue's competitiveness and broad applicability for InfiniBand and other network architectures.

1.3 Thesis Organization

This thesis, especially Chapters 4 – 6, focuses on flow-oblivious, static, destination-based, unicast routing for supercomputers, and is based on the following three peer-reviewed publications which have been reformatted and/or partially rewritten, and the author's contributions are as follows:

- J. Domke, T. Hoefler, and S. Matsuoka. „Fail-in-Place Network Design: Interaction between Topology, Routing Algorithm and Failures". In: *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*. SC '14. New Orleans, LA, USA: IEEE Press, Nov. 2014, pp. 597–608. ISBN: 978-1-4799-5500-8. DOI: `10.1109/SC.2014.54`

  The author of this thesis is the main contributor to this publication. In particular, he performed the analysis of failure data, and designed and implemented the simulation framework. Subsequent simulations, analysis of the results, and preparation of the manuscript were done by the author.

- J. Domke and T. Hoefler. „Scheduling-Aware Routing for Supercomputers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, 13:1–13:12. ISBN: 978-1-4673-8815-3. URL: `http://dl.acm.org/citation.cfm?id=3014904.3014922`

  The author is the main contributor to this publication. Besides identifying the disadvantages of the current state-of-the-art HPC routing methods, he was responsible for algorithm and protocol design (including their implementations) to overcome the challenges. The tasks of benchmarking existing systems and conducting wide-ranging simulations with subsequent data analysis, as well as drafting the publication, were carried out by the author.

- J. Domke, T. Hoefler, and S. Matsuoka. „Routing on the Dependency Graph: A New Approach to Deadlock-Free High-Performance Routing“. In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '16. New York, NY, USA: ACM, 2016, pp. 3–14. ISBN: 978-1-4503-4314-5. DOI: `10.1145/2907294.2907313`
  The author is the main contributor to this published work. Both, the new conceptual approach for solving the deadlock problem and its model implementation, were contributed by the author. Additionally, he analyzed the results of the executed simulations, proved mathematical correctness and characteristics of the routing function, and prepared the manuscript for publication.

The remaining chapters comprise the following: Chapter 2 reviews the state-of-the-art with respect to network simulation frameworks and routing algorithms. Especially, fault-tolerant and deadlock-free routing algorithms are of interest. Chapter 3 lays the groundwork for the following chapters by providing network-related and routing-related definitions, by introducing numerous network topologies, and by illustrating the inner workings of various routing algorithms, which will be used for later comparisons. Thereafter, the three Chapters 4, 5, and 6, outline the main contributions in the field of high-speed interconnects by: (1) showing that fail-in-place network operation is feasible and the corresponding throughput degradation manageable, (2) improving the utilization and communication efficiency through scheduling-aware routing, and (3) developing an invariably working deadlock-free routing algorithm. In conclusion, Chapter 7 summarizes the thesis and provides the prospects for future work.

2 Related Work

The following two sections review the related work, relevant to this thesis, in the research domain of high-performance interconnection networks. Section 2.1 outlines state-of-the-art interconnection architectures, topologies commonly used in HPC systems, management and resiliency considerations, as well as simulation frameworks to analyze the effectiveness of the interconnect. Subsequently, Section 2.2 gives a summary of the state-of-the-art deterministic routing algorithms tailored for the application in supercomputers or for the usage in related network architectures, such as Network-on-Chip. Especially, the algorithm's characteristics with respect to achievable latency, throughput, and deadlock-freedom, as well as applicability to current real-world architectures, are of interest. The Table 2.1, provided at the end of Section 2.2, summarizes the attributes of a subset of these algorithms. Readers unfamiliar with the basics of HPC interconnects and routing algorithms are encouraged to review Section 3.1 prior to continuing with this section.

2.1 State-of-the-Art for High-Performance Interconnect Designs

The interconnection networks of the supercomputers in the TOP500 list of November 2016 are essentially dominated by seven different network architectures, e.g., InfiniBand, Gigabit Ethernet, and Intel's Omni-Path, comprising a total of 423 systems. The remaining systems are connected by either IBM's interconnect for the Blue Gene systems [Adi+05; Che+12], Cray's 3-dimensional tori-based Gemini networks and more recent Aries networks [ARK10; Alv+12], the TH Express interconnect used in MilkyWay 2 systems [Pan+14], or Fujitsu's Tofu interconnect [Aji+12a]. Hence, the prevalent topologies used by these supercomputers are fat-trees [PV97] (occasionally deployed as dual-rail fat-tree [Hat11] or tapered fat-tree [Leó+16]), 5-dimensional or 6-dimensional tori in case of Blue Gene systems or Tofu-based systems, respectively, and dragonfly topologies in case of Aries-based HPC systems. Due to the scale of these production systems, in terms of compute node count, fault-free and regular topologies are an exception rather than the norm [SG10], because both network and compute hardware fails frequently.

2.1.1 Resiliency of Networks, Topologies, and Routings

Resiliency of network topologies has been studied extensively in the past. Xu et al. [Xu10] analyzed numerous interconnection topologies, such as De Bruijn graphs, Kautz graphs, meshes, and butterfly networks. However, an analysis of routing algorithms has been done based on edge forwarding index, but not based on actual performance delivered for a traffic pattern. Another comprehensive survey of network fault tolerance was performed by Adams et al. [AAS87]. The authors describe methods to increase fault tolerance through hardware modification of existing topologies, such as adding additional links, ports, or switches. Other attempts to enhance the fault tolerance properties of the network have been performed analytically. One example is the multi-switch fault-tolerant modified four tree network which aims to improve the single-switch fault-tolerant four tree network [SB02]. Another example is the attempt to solve the problem heuristically with genetic algorithms [Szl06].

Co-design strategies have been proposed, such as the F10 network [Liu+13], where the interconnection topology is optimized for the routing algorithm and vice versa. This includes an increase in alternative paths using an AB Fat-Tree instead of an ordinary fat-tree and includes enhanced fault recovery properties through local rerouting. The routing algorithm for F10, besides optimizing for global load balancing, manages the failure detection and propagates this information to a subset of the nearest neighbors of the failure. This subset is smaller for F10 than in an ordinary fat-tree allowing for fast recovery and therefore decreases the potential for the packet loss.

Okorafor et al. [OL05] proposed a fault-tolerant routing algorithm for an optical interconnect, which assumes a fail-in-place 3-dimensional mesh topology and enhances the throughput. The IBM Intelligent Bricks project [Fle+06] investigates the feasibility of building fail-in-place storage systems and includes considerations about the performance loss in the network due to failed bricks. The throughput degradation in the 6x6x6 3-dimensional torus is simulated up to 60% failed bricks. However, no other routing algorithm besides the IceCube mesh routing has been tested.

Fault-tolerant topology-aware routing algorithms have been developed and studied in the past, especially for meshes and tori topologies and in the field of Network-on-Chip, e.g., [Suh+95; DYL02; RFR09; Man+11; AA13], but the usability of these algorithms for other topologies has not been tested. Flich et al. [Fli+12] performed a survey of topology-agnostic routing algorithms and compared them not only analytically, but also evaluated their performance with a flit-level simulator, which realizes its message forwarding based on the smallest logical unit of network information, called flit, required for flow control [DT03, 12.1]. The simulations have been conducted on small 2-dimensional mesh and tori networks, up to a size of 16x8. Additionally to these regular topologies, faulty meshes and tori with an injected number of link failures of 1%, 3%, and 5% were investigated.

2.1.2 HPC Routing Strategies and Network Management

The state-of-the-art routing method for supercomputers is static and flow-oblivious routing, where the packet route is uniquely defined by the source-destination pair, e.g., [Aji+12b; Inf15], but a few alternatives, such as adaptive routing strategies [Ari+10; Alv+12; Bir+15], exist. Switches of adaptive routed networks are capable of adjusting the path towards a destination dynamically in response to changing local or global conditions, such as link load [DT03, p. 189]. Research has shown that oblivious routing can result in local congestion, which increases application runtime, when the routes and communication pattern does not match properly [HSL08]. Yet, its simplicity and practicality makes it the most-used routing mechanism, for example, in all InfiniBand and Ethernet networks, which combined provide ≈58% performance of all systems in the TOP500 list. The objective of these flow-oblivious routings, such as Up*/Down* [Sch+91] and fat-tree routing [Zah+10], or the deadlock-free single-source shortest-path routing algorithm (DFSSSP) [DHN11], is to distribute the routes evenly across the interconnection network. However, this static and global balancing approach is only effective when the system is used by a single parallel application.

Optimizing communication performance for multi-user HPC system has been studied in the past resulting in many different approaches. Mellanox's proprietary traffic-aware routing, short TARA [Mel16], monitors active applications and the network port counters, and attempts path adjustments when congestion is identified. However, TARA is a reactive optimization and depends on the time to collect and analyze all port counters. Furthermore, TARA's ability to enforce property preserving network updates of the forwarding rules is unknown, hence out-of-order packet delivery might be possible.

Application-aware routings, e.g., [ZL92; Kin+09], have been developed, but require detailed knowledge about traffic patterns and injection rates of each application. Additionally, this optimization approach has no notion of a timely behavior of the applications, and might assign too few links to each applications assuming they communicate simultaneously.

Optimizing the job-to-node assignment via the batch system, while considering the network topology [YCM06; HS11; BDM15], is feasible for regular networks, but assumes an idealized routing or ignores it completely. Performing a routing-aware job-to-node mapping [ATB14] by the batch job scheduler has been proposed as alternative. However, the computational complexity to solve the mapping problem is usually infeasible for an online scheduling of batch jobs, or fast and imprecise heuristics need to be used. In addition, this job-to-node mapping might increase the system's fragmentation, or it is hindered by the logical separation of the system through batch queues.

Changing the forwarding rules of the entire supercomputer, either in reaction to a hardware failure or due to desired optimizations, can violate certain network properties, such as in-order delivery. Multiple network update protocols, tackling these problems, have been developed in the past, but the solutions are usually specific to certain technologies and their properties, such as for the broader gateway protocol (BGP) [Fra+07; RZC11] or for software-defined networking (SDN) [Rei+12; McC+15], to name but a few. Yet, no property preserving network update protocol tailored to the conditions and requirements found in lossless InfiniBand fabrics exists, which is the focus network architecture of this thesis.

2.1.3 Simulation Frameworks for Network Analyses

Academia developed various discrete event simulation frameworks to model network protocols or to design and gauge network architectures, such as ns-3 [YHZ09], OMNeT++ [VH08], and ROSS [CBP00], to name but a few. Subsequently, researchers utilize these frameworks to construct modules for individual architectures or degrees of accuracy. For example, a flit-level accurate simulation model for InfiniBand [Ope07], developed by Mellanox and based on OMNeT++, is used to analyze and improve congestion control in IB networks [Rei+11]. Similarly, Yebenes et al. [Yeb+13] tried to implement an efficient packet-level IB simulator for very large topologies used in exascale supercomputers based on OMNeT++, but only showed results for 28x28 meshes and tori. Alternatively, less accurate, but highly efficient packet-level simulations are enabled by the CODES network and storage simulator (based on ROSS) [Cop+11]. Multiple regular topologies, such as (multi-rail) fat-trees using adaptive and deterministic routings [Liu+15; Wol+17], Slim Flies [BH14; Wol+16], or tori [Mub+14] with millions of compute nodes, have been studied using the CODES simulator. Furthermore, many scientists implemented their own simulator, e.g., cycle-accurate interconnection simulator for Network-on-Chip, called BookSim [Jia+13], or the IB simulator used by Sancho et al. [San+02].

The OMNEST framework, a non-commercial version of OMNeT++, is used by Denzel et al. [Den+08; BRM12] to design a parallel network simulator, interfacing with Dimemas and Paraver, which is able to read and replay communication traces of MPI applications. Despite being a flit-level simulator, their Venus toolchain is not tailored to a specific interconnection technology, such as InfiniBand. Both, the simulated regular topologies with up to ≈4,000 compute nodes, i.e., 2/3-dimensional mesh/torus, hierarchical full mesh, and a fat-tree, and the used forwarding rules for these networks are generated by tools comprised in the Venus toolchain. Multiple routing strategies, such as random routing, source routing, and distributed routing, have been implemented.

Chen et al. [Che+16] compares state-of-the-art and emerging topologies, e.g., single-rail 3-level fat-trees, Mellanox's Dragonfly+, and Hewlett Packard's HyperX topology [Ahn+09], etc. These network topologies are validated by means of synthetic workloads, such as nearest neighbor and all-to-all traffic patterns, and realistic application communication patterns collected for selected mini-apps. Despite assuming different routing strategies (deterministic shortest-path routing, random routing, and adaptive and Valiant-like [DT03], the authors ignored the influence of network component failures.

Jain et al. [Jai+16] evaluates the achievable performance of multi-application workloads simultaneously running on a 5-dimensional torus, dragonfly, and fa-tree topology, respectively. The authors combined the CODES framework with BigSim [Zhe+05], which allows the simulations to relay the exact communication pattern of the individual applications, as well as to model their time spend in computation phases. The implemented adaptive routing algorithms have been tailored for the specific topologies, which prevents comparisons. Furthermore, limiting the analysis to certain applications only, and limiting the switch radix to 13 ports for the torus, while allowing the fat-tree to utilize a radix of 72, might not result in generally valid conclusions with respect to the available design space for interconnection networks.

2.2 State-of-the-Art for Deadlock-free Routing Approaches

As outlined in Chapter 1, avoiding deadlocks in the interconnection network, induced by the routing algorithm, is one of the most crucial property of these algorithms or the underlying topology, respectively. However, not all routing algorithms, such as MinHop [Mel13] or the single-source shortest-path routing [HSL09], take deadlocks into consideration. Therefore, methods for deadlock prevention, avoidance, and recovery for networks have been studied extensively in the past, classifiable into four categories:

1. Solving deadlocks through architecture features,
2. Analytical deadlock-prevention, either through appropriate topology design or routing,
3. Deadlock-preventing routings using virtual channels, and
4. Alternative solutions, usually inapplicable to existing architectures.

For example, for routing algorithms which basically ignore the problem, the interconnection architecture defines a packet lifetime, e.g., InfiniBand [Inf15, 12.9.8.4], as a fall back for categories 1. Clearly, the disadvantage of this approach is that it only resolves a single deadlock, which might form again and again, and decreases the achievable throughput. Furthermore, it complicates the design of networking hardware, such as switches and routes, which have to monitor and enforce packet lifetimes.

Representative routings of category 2 are essentially derivatives of Up*/Down* routing [Sch+91], which prohibit certain turns within the topology to avoid the creation of a routing deadlock. Alternatively, the correct combination of chosen topology and (usually) deadlock-prone routing, such as any loop-free routing on binary trees or dimension order routing on n-dimensional meshes [RR10], ensures deadlock-free transmission in the network. However, prohibiting turns, and therefore restricting the number of possible routes, increases path lengths, hence latency, and overloads nodes resulting in diminished throughput [SRD04]. The algorithms of the third category are relying on virtual channels [DS87] or virtual layers [Lys+06], respectively, and are assigning paths (partially or fully) to different virtual channels to create an acyclic and routing-induced channel dependency graph. While overcoming the path length and throughput constraints of the previous category, this approach requires more complex switch and router designs [San+02], and is potentially unbound in the number of required virtual channels [DHN11; Fli+12] to achieve deadlock-freedom. However, current interconnection hardware only supports a limited number of virtual channels (VCs). For instance, obtainable InfiniBand network components maximally support eight VCs for data traffic, called virtual lanes, while the InfiniBand standard caps the theoretically supported number to 15 data VCs and one VC for subnet management traffic [Inf15, 7.6]. The following Sections 2.2.1 and 2.2.2, as well as Table 2.1, review and summarize a subset of the wide variety of (deadlock-free) deterministic topology-aware and topology-agnostic routing algorithms, respectively.

Category 4 comprises multiple different approaches to enforce deadlock-freedom within the interconnection network. Examples are the “controller principle” [Tou80], where a global entity oversees all network components, and controls packet injection/consumption rates and the forwarding of individual packets to avoid forming a deadlock. Obviously, this approach is unsuitable for modern large-scale interconnects, as well as the latency requirements in high-performance computing. Other approaches for deadlock-freedom are usually inapplicable due to missing features of state-of-the-art network architectures, such as “destination renaming” [LFD01] which easily exceeds the available memory to store forwarding tables in the switches, or “bubble flow control” [Pue+99; WCP13] which aims at controlling the filling level of port buffers to ensure that at least on packet can be forwarded at any given time.

2.2.1 Topology-aware Routing Algorithms

Many of the proposed topologies for supercomputers and other interconnected systems come with a fitting (deadlock-free) topology-aware deterministic routing. For example, Zahavi et al. [Zah10] proposed D-Mod-K routing for parallel port fat-trees. Kim et al. [KBD07] suggested dimension-order routing as a possibility for flattened butterfly NoC topologies. The randomized partially-minimal routing has been tailored for 3-dimensional meshes [RL08] to improve the worst-case throughput, but requires 2 virtual channels for deadlock-freedom. The three topology-aware routings investigated in this thesis, i.e., dimension-order routing (DOR) [RR10] for meshes, Torus-2QoS routing [Mel13], and fat-tree routing [Zah+10], will be explained further in Section 3.3. Generally, the disadvantage of topology-aware routings is their limited fault-tolerance, and therefore limited applicability to large-scale networks, where a number of faults impedes the algorithm's ability to identify the remaining graph as supported topology. However, most these algorithms calculate the deadlock-free forwarding rules for the network faster in comparison to topology-agnostic algorithms, while providing highly optimized routing for low latency and high message throughput.

2.2.2 Topology-agnostic Routing Algorithms

A well-known example for algorithms avoiding to create cycles in the channel dependency graph (CDG) is the Up*/Down* routing [Sch+91]. Up*/Down* prohibits a route to use an "up" direction after a "down" directions. This approach does not necessarily use shortest paths or load-balances routes efficiently. Indeed, the root often becomes a bottleneck in practice. The algorithms UD_DFS routing [SRD00a], L-turn routing [Koi+01] and segment-based routing (SR) [Mej+06] are based on Up*/Down* and try to reduce or balance the routing restrictions to increase the path balancing across the network. For network technologies where the next channel in each routing step is chosen based on the source and destination node the Multiple Up*/Down* routing (MUD) [LS01; Fli+02] can increase the path balancing. For similar network technologies without virtual channel support the Tree-turn routing [ZC12] or FX routing [SRD00b] can be used. For example, Tree-turn adds two more directions to the four directions used by L-turn routing, which reduces the number of prohibited turns further to increase the balancing.

Another set of algorithms breaks cycles in the channel dependency graph with virtual channels. The destination-based routing algorithms DFSSSP [DHN11] and LASH [SLT02] operate similarly in terms of breaking the cycles, i.e., searching for cycles in the CDG and moving individual paths to other virtual layers. Unfortunately, both algorithms might suffer from a limited number of available virtual channels, therefore LASH-TOR [Ske+04] enhanced LASH routing to use Up*/Down* in the last virtual layer if the routes in this layer form a cycle which cannot be resolved. This can result in multiple outgoing ports at a switch for a single destination, hence LASH-TOR is not a destination-based routing in the general case, and therefore inapplicable to most network architectures, especially InfiniBand.

Kinsy et al. [Kin+09] proposed two application-aware routings, called bandwidth-sensitive oblivious routing (with minimal routes) or BSOR(M). BSOR operates within the CDG to calculate the routes, while randomly deleting edges from the CDG to form an acyclic channel dependency graph. The algorithm solves a multi-commodity flow problem, based on the demands of the application, with a mixed-integer linear programming algorithm for small networks. For large networks, BSOR uses Dijkstra's algorithm as a heuristic on a weighted and acyclic channel dependency graph for each source/destination pair to balance the application traffic. In contrast, BSORM routing calculates the routes within the network and breaks cycles afterwards, resembling the method of DFSSSP and LASH. BSOR(M) is designed for network technologies with forwarding based on source and destination, and therefore are inapplicable to InfiniBand for example. The same holds for smart routing [CKR96]. The approach of smart routing is to calculate the shortest paths and investigate the routing-induced channel dependency graph for cycles, while storing which path induced which edge in the CDG. A cycle search in the CDG subsequently cuts the edges of a cycle which minimizes the average path length after recalculating the paths inducing this edge. While smart routing can be used for technologies without virtual channels, the computational cost, which is $\mathcal{O}\big((\#\text{switch})^9\big)$, is to high for a practical use in large-scale networks.

Table 2.1: Qualitative comparison of existing topology-aware and topology-agnostic routing algorithms for supercomputers; Highlighted in red: limitations and shortcomings

Routing	Network Topology	Latency	Throughput	Deadlock-Freedom	#VC needed	Fault-Tolerant	Time Complexity[#]
topology-aware							
DOR	meshes	+	+	yes	1	no	N/A
Torus-2QoS	2D/3D meshes/tori	+	++	yes	≥ 2	limited	N/A
Fat-Tree	k-ary n-trees/XGFTs	+	++	yes	1	limited	N/A
topology-agnostic							
MinHop	arbitrary	+	+	no	1	yes	$\mathcal{O}(\lvert N\rvert \cdot \lvert C\rvert)$
Up*/Down*	arbitrary	--	--	yes	1	yes	$\mathcal{O}(\lvert N\rvert \cdot \lvert C\rvert)$
MUD	arbitrary	-	-	yes	≥ 2	yes	$\mathcal{O}(\lvert N\rvert \cdot \lvert C\rvert)$
SSSP	arbitrary	+	++	no	1	yes	$\mathcal{O}(\lvert N\rvert^2 \cdot \log \lvert N\rvert)$
DFSSSP	arbitrary	+	++	yes[*]	≥ 1	yes	$\mathcal{O}(\lvert N\rvert^2 \cdot \log \lvert N\rvert)$
L-turn	arbitrary	-	-	yes	1	yes	$\mathcal{O}(\lvert N\rvert^3)$
LASH	arbitrary	+	-	yes[*]	≥ 1	yes	$\mathcal{O}(\lvert N\rvert^3)$
LASH-TOR	arbitrary[†]	-	-	yes	≥ 1	yes	$\mathcal{O}(\lvert N\rvert^3)$
SR	arbitrary	-	-	yes	1	yes	$\mathcal{O}(\lvert N\rvert^3)$
Smart	arbitrary	-	+	yes	1	yes	$\mathcal{O}(\lvert N\rvert^9)$
BSOR(M)	arbitrary[†]	+	++[‡]	yes	≥ 1	yes	N/A
versatile, topology-agnostic, high-performance routing							
Novel approach[$]	arbitrary	-/+	+/++	yes	≥ 1	yes	$\leq \mathcal{O}(\lvert N\rvert^3)$

[†] Inapplicable to current interconnection hardware, such as InfiniBand, because destination-based routing criterion might be violated.

[‡] Requires knowledge about the bandwidth demands of the application(s) to optimize the forwarding rules.

[*] Can require more virtual channels than there are available and hence renders the algorithm unusable to calculate deadlock-free routes.

[#] Time complexity to calculate all source-destination paths for the entire network $I = G(N, C)$ consisting of $\lvert N\rvert$ nodes interconnected by $\lvert C\rvert$ links [DHN11; Fli+12].

[$] This approach will be the subject of Chapter 6 and can be further tuned by applying the knowledge gained in Chapter 5, as outlined in the conclusion of this thesis.

3 Background, Assumptions, and Definitions

This chapter formalizes network-related terms, which will be used throughout the thesis to accurately design, enhance, and compare network topologies and routing algorithms. Section 3.1 defines the terms interconnection network, path, and routing function and introduces common metrics. Subsequently, Section 3.2 gives an overview of a subset of network topologies, which have been proposed for the usage in HPC systems. Furthermore, a number of modern real-world installations and their topologies will be presented. The basis operating principles of a selection of routing algorithms will be given in Section 3.3. These algorithms are used throughout the world in production supercomputers and therefore serve as reliable baselines for later comparisons.

3.1 Interconnection Networks and Routing Algorithms

First, and foremost, an exact characterization of the network is needed when working on network topology design, as described in Section 3.1.1. This graph-based specification, see Definition 3.1, builds the basis for subsequent terms, such as path or route and derived routing functions.

3.1.1 Network-related Definitions

Within the scope of interconnection networks for high-performance computers [DYL02], the assumption is that a network is a multigraph where each pair of network devices is connected by one or more full-duplex channels (or links). These full-duplex links can be simultaneously used in opposing directions without interference or throughput degradation. Furthermore, the link capacity, i.e., the maximum achievable throughput, in the network is assumed to be uniform and constant over time.

Definition 3.1 (Interconnection Network)**.** *An* ***interconnection network*** $I := G(N,C)$ *is a connected multigraph with the node set* N *and multiset* C *of directed (multi-)channels. Nodes* $n_x \in N$ *are called a* ***terminal****, if and only if there exists exactly one* n_y *with* $(n_y,n_x) \in C$*, otherwise* n_x *is called a* ***switch****. Furthermore, let* $C^* \subseteq C$ *be the largest possible subset, with* $\forall c_{n_x,n_y} \in C^* : n_x, n_y$ *being switches* $\in N$*, denoting the set of inter-switch links of network* I*.*

If exact device information is required, e.g., for figures, then ordered pairs (n_x,n_y), or c_{n_x,n_y} for short, are used, otherwise a simpler c_i notation denotes channels. In the following, S is used to denote the set of switches and the set T is used for terminals, hence $N = S \cup T$ with $S \cap T = \emptyset$. Figure 3.1 shows an example network consisting of five switches and no terminal nodes. For a more precise delimitation, terminals

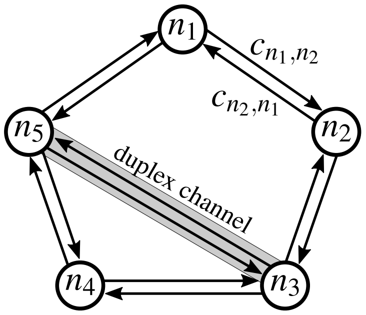


Figure 3.1: 5-node interconnection network $I = G(N,C)$; $N = S \cup T$ with $|S| = 5$, $|T| = 0$; Full-duplex channels C allow bi-directional, simultaneous, non-interfering transmission

can be further subdivided into *compute nodes*, executing scientific applications in the supercomputer, and *supplementary nodes*, responsible for storage, administration, etc. Nodes in the network communicate via *messages* of arbitrary length, which are transferred as a payload of one or more *network packets* from a sending node to a receiving node. The packet routes, see Definition 3.2, for all nodes to all other nodes are defined by a routing function, see Definition 3.4, within the multigraph representing the network. While the mathematical concept of cyclic paths is valid, for the remainder of the thesis the absence of cycles, or so called *routing loops* [SKC14, pp. 230-231], is required, see Definitions 3.2 and 3.4. Otherwise, since network switches or routers commonly cannot distinguish between the first or additional passes of a packet, a packet might be trapped in an infinite forwarding cycle through the network.

Definition 3.2 (Cycle-free Route). *A* ***route (or path)*** P_{n_x,n_y} *of length* $h \geq 1$ *from node* n_x *to* n_y *in the network* $I = G(N,C)$ *is defined as a sequence of channels* $(c_1,\ldots,c_h) =: P_{n_x,n_y}$ *under the condition that* $\{c_1,\ldots,c_h\} \subseteq C$, $c_1 := (n_x,\cdot)$, $c_h := (\cdot,n_y)$, *and if* $c_q = (\cdot,n_z)$ *then* $c_{q+1} = (n_z,\cdot)$ *for all* $1 < q < h$. *The path* P_{n_x,n_y} *is considered to be* ***cycle-free****, if for all* $1 \leq p,q \leq h$, *with* $p \neq q$ *and* $c_p = (\cdot,n_u)$, $c_q = (\cdot,n_v)$, *it holds that* $n_u \neq n_v$. *Let* ${}^{s}P_{n_x,n_y}$ *denote a shortest path from* n_x *to* n_y.

Depending on the routing functions, a categorization in shortest-path routing algorithms, i.e., the algorithm calculates ${}^{s}P_{n_x,n_y}$ for all pairs of nodes, and non-shortest-path routings is possible, as outlined in Chapter 2. Many of the proposed and state-of-the-art routing algorithms optimize for shortest paths to minimize the latency observed by packets. However, the shortest-path criteria enforces limitations with respect to deadlock-freedom, which will be further analyzed in Chapter 6.

For a fair comparability of the different topologies in the following chapters, especially Chapter 4, it is frequently essential to add redundant links in the network to utilize all available switch ports. Hence, the regular graph-based topology is augmented to an undirected multigraph. As a result, the number of hosts, switches, and links can be balanced across the different investigated topologies. The link redundancy $r_{C'}$, see Definition 3.3, which increases the edge-connectivity, but not the vertex-connectivity, is defined as the smallest number of parallel links between two adjacent switches. The two network metrics vertex-connectivity and edge-connectivity will be further explained in Section 3.1.3.

Definition 3.3 (Network Link Redundancy)**.** *Assuming an interconnection network* $I = G(N,C)$*, then the augmented interconnection network with redundant channels or links is given by* $I' := G(N,C')$*, with* $|\{c'_{n_x,n_y} \in C'\}| \geq |\{c_{n_x,n_y} \in C\}|$ *for all* $n_x, n_y \in S$*, and the (minimal)* ***link redundancy*** $r_{C'}$ *for the topology is defined by*

$$r_{C'} := \min_{n_x,n_y \in S} \left\lfloor \frac{|\{c'_{n_x,n_y} \in C'\}|}{|\{c_{n_x,n_y} \in C\}|} \right\rfloor$$

Unless otherwise stated in later chapters, the interconnection networks and topologies defined in Section 3.2 have not been altered from their original specification. Hence, the default value for all these topologies is a node/link redundancy of $r = 1$.

3.1.2 Routing-related Definitions

Generally speaking, the routing configuration is part of the switching rules for a network, which define a set of valid forwarding rules and packet modifications along the path to the destination. The routing function R, as defined hereafter, specifies the paths from all nodes to all other nodes in the network I.

Definition 3.4 (Destination-based and Cycle-free Routing)**.** *A* ***routing function*** $R : C \times N \to C$ *for a network* $I := G(N,C)$ *assigns the next channel* c_{q+1} *of the route depending on the current channel* c_q *and the destination node* n_y*. The ports of the switch, connecting these two channels, are categorized in* ***ingress port*** *for* c_q *and* ***egress port*** *for* c_{q+1}*, respectively. For multigraphs, the assumption is that the next channel* c_{q+1} *is unique among the existing parallel channels of a multi-channel. Furthermore, a routing R is considered to be* ***destination-based*** *if the channel* c_{q+1} *is unique (denoted by the* $\exists!$ *sign) at each node, i.e.,* $\forall n_x \in N \; \exists! c \in C : R\big((\cdot, n_x), n_y\big) = c$*. The routing R is* ***cycle-free*** *if all paths induced by R are cycle-free.*

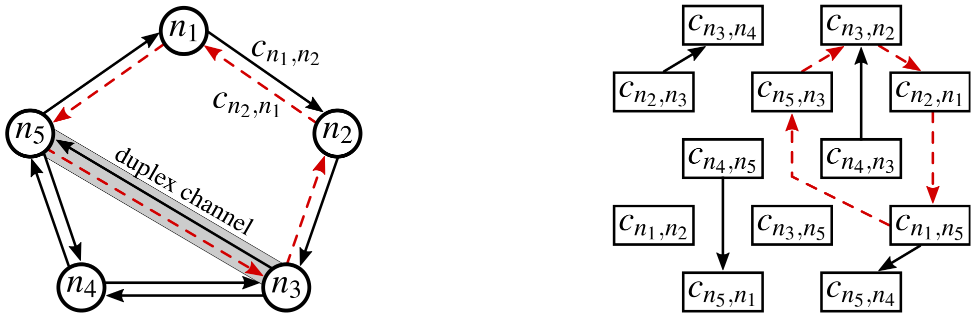
The forgoing definition of the routing function R is required to express channel dependencies later on, although theoretically it is equivalent to $R' : N \times N \to C$.

Modern high-performance interconnect technologies, such as InfiniBand or Intel's Omni-Path, encapsulate these forwarding rules for each network switch in so-called linear forwarding tables (LFTs):

Definition 3.5 (Linear Forwarding Tables)**.** *The* ***linear forwarding table*** *of a network switch* $s \in S$ *is a per-packet lookup table derived from the routing function R. The destination of a packet is the index for the array and the switch's egress port is stored in it, i.e.,* $\text{egress port} = LFT[n_y]$ *defined by* $R\big((\cdot, s), n_y\big)$*.*

These lookup tables are usually programmed by a routing algorithm, either executed by a centralized entity, such as InfiniBand's subnet manager [Mel13], or executed on each switch individually, resulting in a distributed routing, such as the Border Gateway Protocol (BGP) [Fra+07]. The complete set of all forwarding tables, regardless of being calculated centrally or in a distributed fashion, is called a (global) network state from here on.

Definition 3.6 (Network State)**.** *The* ***network state*** *is the set of all forwarding tables of network switches given by a valid routing function R for an interconnection network I.*



(a) Ring with shortcut **(b)** Corresponding (cyclic) CDG

Figure 3.2: Using a shortest-path, counter-clockwise routing for network I (3.2a) induces the channel dependency graph D (3.2b); Dashed channel dependencies in D form a potential deadlock, induced by paths of length $h = 2$ (compare to Definition 3.2) which use dashed channels of network I

A routing function R for a given network I is considered to be valid, if and only if the routing has these three properties: *cycle-free*, *destination-based*, and *deadlock-free*, so that the routing is applicable to InfiniBand, Ethernet, and many Network-on-Chip designs. While the cycle-free and destination-based properties follow from Definition 3.4, deadlock-freedom cannot be accomplished so easily. Especially, the InfiniBand standard [Inf15, C14-62.1.2] demands the use of deadlock-free routing algorithms. Dally et al. [DS87] postulate Theorem 3.1, which sets the necessary and sufficient condition for the deadlock-freedom of a destination-based routing function.

Theorem 3.1. *A routing function R for an interconnection network I is deadlock-free if and only if there are no cycles in its corresponding channel dependency graph.*

The corresponding channel dependency graph (CDG) is induced by the routing function R, or the network state, respectively, for a given network I as follows:

Definition 3.7 (Channel Dependency Graph)**.** *The **channel dependency graph** for a given network I and routing function R is defined as a directed graph $D := G(C,E)$ whose node set consists of the channel set of I. The edge set of D, denoted by ordered channel pairs $(\cdot,\cdot)$, is induced by the routing function, i.e., $(c_p, c_q) \in E$ if and only if $\exists n_y \in N : R(c_p, n_y) = c_q$.*

Figure 3.2b shows the channel dependency graph for the network shown in Figure 3.2a assuming that a shortest-path, counter-clockwise routing function is used. For instance, the path $n_2 \rightarrow n_5$ via n_1 induces the rightmost channel dependency, i.e., the edge between c_{n_2,n_1} and c_{n_1,n_5} in Figure 3.2b. The "shortest-path" property is considered the primary path selection criterion in this example.

Dally et al. [DS87] also introduced the concept of *virtual channels* (VCs) to solve the deadlock problem for a limited set of network topologies. Actual switching hardware implements these virtual channels $\tilde{c}$ with separated sets of buffers within the same port or channel $c \in C$, respectively. One method for breaking a cyclic channel dependency graph is to conditionally switch between virtual channels along a route, so called virtual channel transition, to obtain a deadlock-free routing. However, not every interconnection technology supports VC transition, despite supporting virtual channels, as mentioned in Section 1.1. An alternative way of achieving deadlock-freedom on these interconnects, such as InfiniBand, is to combine the VCs into virtual layers [LS01]. Routes are then assigned to individual layers, see Definition 3.8, so that all routes within one layer induce an acyclic CDG. Hence, all layers are acyclic and therefore the routing is deadlock-free.

Definition 3.8 (Virtual Layer)**.** *Assume each network channel* $c \in C$ *of* I *can be split into* k *virtual channels* $\{\tilde{c}^1, \dots, \tilde{c}^k\}$, *with* $k \in \mathbb{N}$, *then the i-th* ***virtual layer*** $L_i := G(N, C_i)$ *of the interconnection network* $I = G(N, C)$ *is determined with* $C_i := \left\{\tilde{c}^i \mid \tilde{c}^i \cong c \text{ for } c \in C\right\}$. *The network* I *and virtual layer* L_i *are identical for* $k = 1$.

In a simplified manner, the concept of virtual channels is comparable to the link redundancy introduced in Definition 3.3, assuming $k = r_{C'}$, without the benefit of an increased bandwidth in the network. However, arbitrarily increasing the link redundancy is unfeasible due to the costs required to build high-radix switches (with radix > 128) [Kim+08]. A vast number of virtual channels is not cost-efficient either, hence high-radix switches are combined with virtual channel support. The technique of assigning routes to individual layers is used by DFSSSP [DHN11] and LASH [SLT02] routing (among others). A key challenge with virtual channels is that algorithms that rely on computing shortest-paths can require more virtual channels than available, see Section 6.3.3 for further details, and an optimal assignment of routes to layers is an NP-complete problem [DHN11]. Thus, potentially rendering these algorithms inapplicable for the given hardware and topology. At the same time fall-back algorithms, such as Up*/Down* or L-turn routing, can adhere to the available hardware, but will compute drastically worse routings. This challenge motivates the design of alternative or novel routing approaches, such as routing on the channel dependency graph presented in Chapter 6.

3.1.3 Topology and Routing Metrics

A wide range of metrics for network topologies exist that are mostly derived from graph-based metrics, such as switch radix, network diameter, or path length. Moreover, the set of metrics derived from the interconnection technology includes link bandwidth, port-to-port latency, bisection bandwidth, etc. These metrics make a statement about idealized conditions while ignoring the used routing algorithm. However, more accurate routing-based metrics, such as the edge forwarding index, see Definition 3.12 or the effective bisection bandwidth [HSL08], usually require extensive simulations or measurements on real systems. This section provides a selection of metrics which are relevant for the analyses conducted in subsequent chapters.

Building resilient and fail-in-place networks requires not only a fault-resilient routing algorithm, see Chapter 4, but also highly connected topologies. One resilience measurement for topologies is the connectivity of the graph $G(N, C)$ representing the network I [Xu10]. The connectivity determines the minimum number of network components which have to be removed in order to disconnect the network, i.e., to arrive at the worst case scenario, but neither does it estimate the average case nor can throughput degradation be derived from it. Let a path P_{n_x,n_y} from n_x to n_y be a sequence of channels $c_i \in C$, see Definition 3.2, connecting a sequence of nodes $(n_x, v_1, \ldots, v_{h-1}, n_y)$, with $v_i \in N$ and $h \geq 1$. The (n_x,n_y)-vertex cut $\kappa(G;n_x,n_y)$ for any non-adjacent pair (n_x,n_y) of nodes in N is equal to the maximum number of pairwise internally vertex-disjoint paths from n_x to n_y. For example, the (n_1,n_4)-vertex cut of the graph shown in Figure 3.2a is two, since only two vertex-disjoint paths exist, i.e., via n_5 or via n_2 and n_3, respectively. Similarly, the (n_x,n_y)-edge cut $\lambda(G;n_x,n_y)$ is equal to the maximum number of pairwise edge-disjoint paths from n_x to n_y, however without the non-adjacency requirement. Hence, $\kappa(G;n_x,n_y)$ or $\lambda(G;n_x,n_y)$ count the minimum number of nodes or channels, respectively, which have to be removed to disconnect both nodes. The vertex-connectivity and edge-connectivity of I can be derived from the vertex/edge cuts by determining the minimum among all cuts:

Definition 3.9 (Connectivity and Edge-Connectivity)**.** *The* ***(vertex-)connectivity*** $\kappa(G)$ *of an interconnection network* $I = G(N,C)$ *is defined by the smallest number of* (n_x,n_y)*-vertex cuts*

$$\kappa(G) = \min\{\kappa(G;n_x,n_y) \mid \forall n_x,n_y \in N \wedge (n_x,n_y) \notin C\}$$

and its ***edge-connectivity*** $\lambda(G)$ *is defined by*

$$\lambda(G) = \min\{\lambda(G;n_x,n_y) \mid \forall n_x,n_y \in N\}$$

Whitney's inequality states that $\kappa(G) \leq \lambda(G)$ for any graph [Xu10, Theorem 1.5.2]. Unless mentioned otherwise, for this metric the interconnection network I is extended to terminal graphs $G(N,C,T)$, where $N \setminus T = S$ represents the set of switches in I and T represents the set of terminal nodes in I. The connectivity analysis is performed on the subgraph $G(S,C^*)$, or otherwise $\kappa(G)$ and $\lambda(G)$ would always be equal to one for these types of connected graphs.

Another important network metric is the observed end-to-end latency. However, this latency varies not only depending on the used routing function, but also depending on the current network load [San+02]. To reduce the degrees of freedom, routing functions are compared by the ideal latency while ignoring any background traffic. This ideal latency can be derived directly from the hop count, or path length respectively, for each routing.

Definition 3.10 (Hop Count)**.** *Assuming a given interconnection network I and routing function R, then the routing-based* ***hop count*** h_R *for a path between* n_x *and* n_y *is the cardinality of the path sequence induced by the routing function:*

$$h_R(P_{n_x,n_y}) := \left|\{\underbrace{(n_x,\cdot)}_{=:c_1}, \underbrace{R(c_1,n_y)}_{=:c_2}, \ldots, \underbrace{R(c_i,n_y)}_{=:c_{i+1}}, \ldots, \underbrace{(\cdot,n_y)}_{=:c_h}\}\right|$$

which is therefore always $h_R(P_{n_x,n_y}) \geq h({}^sP_{n_x,n_y})$ *for any pair of nodes.*

While hop count accurately quantifies the peak observable latency for communication peers, it cannot be used for more complex communication patterns, such as all-to-all, bit-reversal, or hot-spot traffic patterns [Fli+12]. Both, the network topology and used routing algorithm, heavily influence the processing time of message bursts, e.g., the send-deterministic communication patterns found in HPC applications [CGS10]. Hence, analyzing the network throughput, see Definition 3.11, provides an instrument for qualitative and quantitative assertions of topology and routing combinations in later chapters.

Definition 3.11 (Network Throughput)**.** *Let t_s and t_e be the times of injection of the first packet into the network and consumption the last packet belonging to a message burst M, respectively, then the routing-depended **network throughput** is*

$$throughput := \frac{\sum\limits_{m \in M} sizeof(m)}{t_e - t_s}$$

the sum of messages sizes in the burst divided by the runtime.

When the exact communication pattern is unknown, then another, more abstract method for routing comparisons relies on counting the number of paths passing through a network node, or channel, respectively. The edge forwarding index of individual ports, see Definition 3.12, or the network itself, estimates a theoretical upper bound for the worst-case congestion observable for any communication pattern [Chu+87; HMS89]. Therefore, the objective of all traffic-oblivious routing functions should be to minimize the network's forwarding index and balance the edge forwarding indices of all ports.

Definition 3.12 (Edge Forwarding Index)**.** *Let $I = G(N,C)$ be the interconnection network and let $R : C \times N \to C$ be the routing function that configures the forwarding tables, then the **edge forwarding index** γ of a channel $c \in C$ is the sum of the routes being forwarded via this channel, i.e.,*

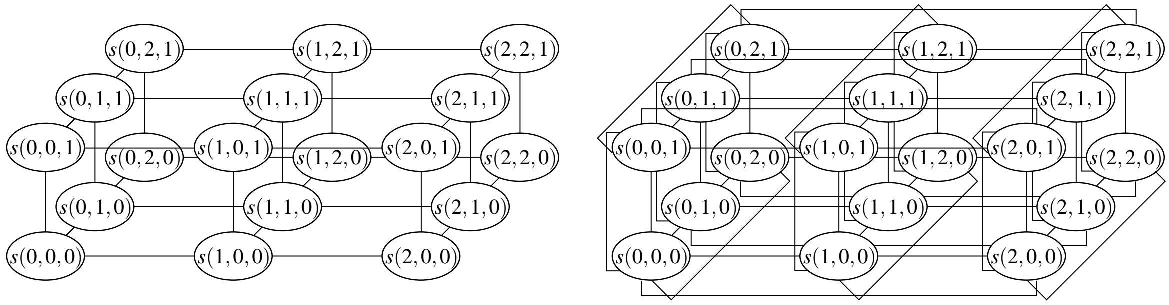
$$\gamma_R(c) := \left|\{P_{n_x,n_y} \mid n_x, n_y \in N \wedge R(\cdot, n_y) = c \in P_{n_x,n_y}\}\right|$$

and accordingly, the edge forwarding index of the network is defined by $\gamma(I,R) := \max\limits_{c \in C} \gamma_R(c)$.

The above introduced metrics, and derivates thereof, will be used for meaningful comparisons in Chapters 4 – 6. Other metrics might provide similar insight with respect to the quality of routing functions, but are beyond the scope of this thesis.

3.2 Common Network Topologies and Production HPC Systems

The most common topologies used to connect the compute nodes in modern large-scale supercomputers are based on fat-trees (Section 3.2.2), or higher-dimensional tori (Section 3.2.1). A few examples are the fat-tree used by the No. 1 supercomputer of the TOP500 list [Str+16] of November 2016. This 40,960-node system, called Sunway TaihuLight, is located at the National Research Center of Parallel Computer



(a) Mesh(3,3,2) topology

(b) Torus(3,3,2) topology

Figure 3.3: Example visualization of a 3-dimensional mesh (3.3a) and torus (3.3b) topology without terminals

Engineering & Technology (NRCPC), China [Don16]. Furthermore, the Oak Ridge National Laboratory (ORNL), USA, is planing the dual-rail fat-tree based Summit HPC system [Oak14]. Last but not least, the K computer at the RIKEN research institution in Kobe, Japan, utilizes a 6-dimensional torus connecting 82,944 compute nodes [Ino10]. However, the TOP500 list also includes other emerging topologies, such as dragonfly-based networks deployed by Cray [Faa+12]. This section provides an overview of common HPC interconnection topologies which will be used in later chapters to compare existing and new routing algorithms.

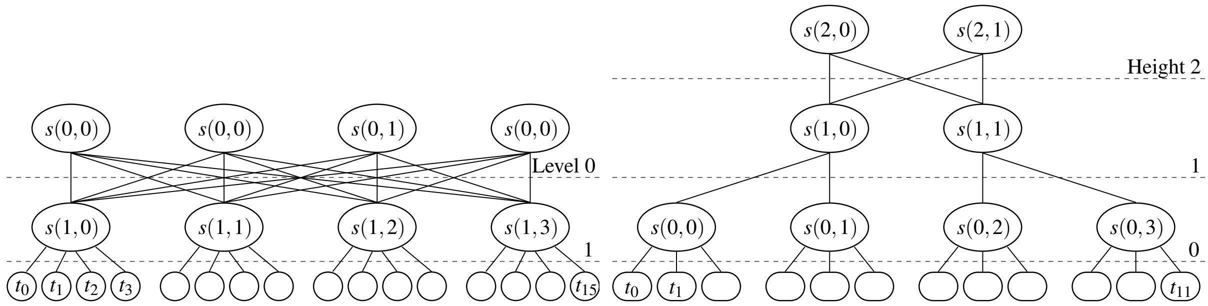
3.2.1 Meshes and Tori

Meshes are primarily utilized to build Network-on-Chip architectures (NoC) [Pal+06; Man+11; Sod+16], while tori have been used to connect IBM Blue Gene systems [Adi+05; Che+12] over the last years. A mesh network $\text{mesh}(d_1,\ldots,d_n)$, with $d_i \geq d_{i+1}$ for $i = 1,\ldots,n$, of dimension n is defined as an n-dimensional matrix, with each switch having at most $2 \cdot n$ adjacent switches, i.e., two switches in each dimension [WC04]. Figure 3.3a illustrates a 3-dimensional mesh with $d_1 = 3$ and $d_2 = d_3 = 2$. The set of adjacent switches N_i^M for switch s_i with dimension indices $i = (i_1,\ldots,i_n)$ is given by

$$\begin{aligned} N_i^M(s_i) = &\big\{s_{j_-} \in S \mid j_- = (i_1,\ldots,i_k - 1,\ldots,i_n) \text{ with } i_k - 1 \geq 1 \;\; \forall k = 1,\ldots,n\big\} \cup \\ &\big\{s_{j_+} \in S \mid j_+ = (i_1,\ldots,i_k + 1,\ldots,i_n) \text{ with } i_k + 1 \leq d_k \;\; \forall k = 1,\ldots,n\big\} \end{aligned} \tag{3.1}$$

Similarly, an n-dimensional $\text{torus}(d_1,\ldots,d_n)$, as shown in Figure 3.3b, is defined as an n-dimensional matrix, where each switch has exactly $2 \cdot n$ neighbors. For tori the boundary constraints are removed and switches are connected in a circular manner and the set of adjacent switches N_i^T is defined as

$$\begin{aligned} N_i^T(s_i) = \big\{s_{j_-}, s_{j_+} \in S \mid\, &j_- = (i_1,\ldots,i_k - 1 \bmod d_k,\ldots,i_n) \\ &\text{and } j_+ = (i_1,\ldots,i_k + 1 \bmod d_k,\ldots,i_n) \;\; \forall k = 1,\ldots,n\big\} \end{aligned} \tag{3.2}$$



(a) Full-bisection 4-ary 2-tree with 16 terminals **(b)** XGFT(2;2,2;1,2) with 12 terminal nodes

Figure 3.4: Example visualization of a k-ary n-tree topology (3.4a) and an extended generalized fat-tree topology (3.4b)

The set of terminals T is equally distributed over the switch set S. Therefore, each switch has either $\left\lfloor \frac{|T|}{|S|} \right\rfloor$ or $\left\lceil \frac{|T|}{|S|} \right\rceil$ attached terminals. Unless mentioned otherwise, the terminal distribution of T is performed similarly for all topologies explained in Sections 3.2.1 – 3.2.4.

3.2.2 Fat-Trees and eXtended Generalized Fat-Trees

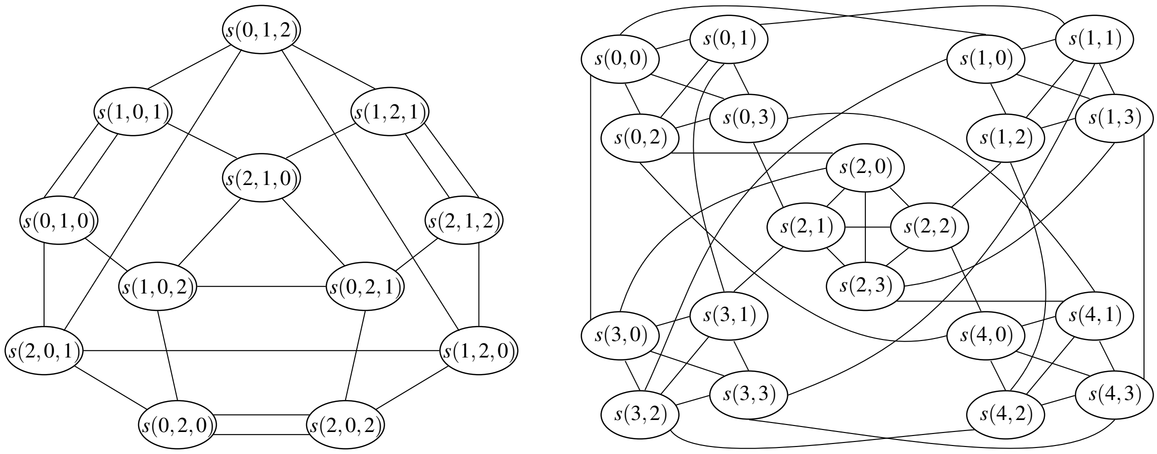
The k-ary n-trees [PV97] are a special form of the fat-tree networks. The parameters k and n define the number of child nodes and the height of the tree, respectively. One characteristic of k-ary n-trees is the number of switches per level, which is equal to k^{n-1}, and therefore constant and independent of the specific level of the tree. Assuming, each switch $s \in S$ is associated with a unique identifier $(\omega_1, \dots, \omega_{n-1}, l)$, with $\omega_i \in \{1, \dots, k\}$ $\forall i$ and level $l \in \{0, \dots, n-1\}$, then there exists a connection between switch $s_i = (\omega_1, \dots, \omega_{n-1}, l)$ and $s_i^* = (\omega_1^*, \dots, \omega_{n-1}^*, l+1)$ if and only if

$$\forall i, 1 \leq i \leq n-2, i \neq l : \omega_i = \omega_i^* \tag{3.3}$$

The switches with level $= 0$ are root switches of the k-ary n-tree. One advantage of the k-ary n-trees is the theoretical full-bisection bandwidth between all nodes in the network.

The most generic and variable form of fat-trees are the extended generalized fat-trees (XGFT) [Öhr+95]. The advantage of an XGFT network is the ability to scale the network size depending on the needs, either with focus on high bisection bandwidth or with focus on reducing the costs for switches and links in the higher levels of the tree. The informal and recursive definition of $\text{XGFT}(h+1, m_1, \dots, m_{h+1}, \omega_1, \dots, \omega_{h+1})$ with height $h+1$ is the composition of m_h copies of the subtree $\text{XGFT}(h, m_1, \dots, m_h, \omega_1, \dots, \omega_h)$ and $\omega_1 \cdot \ldots \cdot \omega_{h+1}$ new root nodes. The original root nodes of these subtrees are then connected to the new root nodes in level $h+1$ to form the full XGFT. Refer to Öhring et al. [Öhr+95] for the formal definition of the extended generalized fat-trees. Figure 3.4 shows the layout of a k-ary n-tree and XGFT topology.

The terminal set T of the network I is evenly distributed among the switches of the lowest level of the tree, i.e., $l = n-1$ for k-ary n-trees and $h = 0$ for XGFTs, respectively.



(a) Kautz$(2,3)$ graph without terminals **(b)** Dragonfly(4,0,2,5) connected by complete graphs

Figure 3.5: Example visualization of a Kautz graph (3.5a) and a dragonfly topology (3.5b)

3.2.3 Kautz Graph

A Kautz(d,k) graph [LLS04] is a directed graph, where each node has d outgoing edges. Non-terminal nodes $s \in S$ of Kautz(d,k) are associated with a Kautz string α as follows

$$\alpha \in \left\{a_1 a_2 \dots a_k \mid a_i \in \{0,\dots,d\} \text{ for } 1 \le i \le k \wedge a_i \neq a_{i+1} \text{ for } 1 \le i < k\right\} \tag{3.4}$$

A switch s_α has an outgoing edge to an adjacent switch s_β if and only if

$$\beta = a_2 \dots a_k b \quad \text{with } b \in \{0,\dots,d\} \wedge b \neq a_k \tag{3.5}$$

The advantages of a Kautz graph are the small diameter and high fault tolerance derived from the degree of d, i.e., two desired properties for HPC interconnects. In case of an undirected Kautz graph, i.e., while using full-duplex links in an interconnection network (see Figure 3.5a for an example), the connectivity is $2 \cdot d$ as shown in Table 3.1. For the remainder of this thesis, the assumption is that full-duplex links are connecting the switches in a Kautz graph. Terminals are connected in the same manner as explained for meshes in Section 3.2.1.

3.2.4 Dragonfly Topologies

Kim et al. [Kim+08] introduced the dragonfly network topology. The design goals are scalability, low diameter for low latency, and reduced cost for the network, which can be accomplished by a hierarchical topology design. A set of switches is combined into a group to build a virtual high-radix switch. The groups are then connected among each other to create the entire network. For scalability and variability reasons both the intra-group network topology and the inter-group network topology are not predefined. The dragonfly(a,p,h,g) network is defined by the number of switches a in each group, the number

Table 3.1: Vertex-connectivity for the investigated network topologies of Sections 3.2.1 – 3.2.4 (except for random and Cascade)

Interconnect Topology	Vertex-Connectivity	Reference
Mesh$(d_1,\dots,d_n)$	n	[RR10]
Torus$(d_1,\dots,d_n)$	$2 \cdot n$	[Öhr+95]
k-ary n-tree	k	[Öhr+95]
XGFT$(h,m_1,\dots,\omega_1,\dots)$	ω_1	[Öhr+95]
Kautz(d,k) as undirected graph	$2 \cdot d$	[HL08, p. 118]
Dragonfly(a,p,h,g) with 1D flattened butterfly	$a+h-1$	[Kim+08]

of inter-group links h per switch, and the number of groups g. The parameter p specifies the number of terminals that are connected to each switch. In the following, 1-dimensional flattened butterfly topologies (i.e., a complete graph) are assumed for the intra-group connectivity as well as for the inter-group connectivity [AK11, pp. 34-37]. Therefore, the number of groups is limited to $a \cdot h + 1$ and the network diameter is five when terminal-to-switch connections are included, otherwise three. For instance, Figure 3.5b illustrates a switch-only dragonfly topology with five groups, $a = 4$ switches, and $h = 2$ inter-group links per switch.

Another form of dragonfly topologies, which refrain from using 1-dimensional flattened butterflies, are the Aries-based Cascade networks designed by Cray, Inc. [Faa+12]. Instead of using a complete graph to connect the switches within each group, Cascade arranges these Aries switches in a 16×6 mesh, i.e., the parameter $a = 96$ is preset for Cascade networks. Switches in this mesh have an all-to-all connectivity along the x-axis, as well as the y-axis, resulting in 20 intra-group links per switch. Furthermore, each switch hosts $p = 4$ terminals. The maximum number of inter-group links per switch is 10, allowing a parameter h of up to 960. Since groups are connected by bundles of four links in an all-to-all manner, the Cascade topology can theoretically scale to a total of 241 groups or 92,544 compute nodes, respectively.

Dragonfly networks are intended to be used with adaptive routing, but a load-balanced and deadlock-free shortest-path routing helps to avoid unnecessary non-minimal adaptive routing decisions. Hence, the investigation and comparison of different static routings for dragonflies is valuable.

With respect to the previously defined connectivity metric for interconnection networks, see Section 3.1.3, Table 3.1 summarizes the vertex-connectivity of the previous outlined topologies. Jellyfish topologies (cf. Section 3.2.5) and Cascade topologies are omitted in consequence of their complex topology structures. However, both the (vertex-)connectivity and edge-connectivity can be computed using Menger's Theorem [Xu10, pp. 30-31], if required. Increasing the vertex-connectivity of an HPC interconnection network for supercomputers is rather costly, e.g., by applying the dual-rail approach and connecting the terminals to both network planes [Wol+17]. The TSUBAME2.0 supercomputer follows this approach, see next Section 3.2.6. However, assuming high-radix switches with empty ports, then increasing the edge-connectivity of the network via the usage of a link redundancy $r_{C'} > 1$ is less expensive. Hence, network resiliency as well as available network bandwidth improves.

3.2.5 Random Topologies

Random and pseudo-random topologies [Koi+03; Koi+12], also called Jellyfish topologies [Sin+12], have been proposed for data centers as well as for HPC interconnects. The construction rule used in this thesis for a random topology is to equally distribute the terminals over the switch set S and create a cyclic graph for S, to ensure initial connectivity. Afterwards, additional links, up to a specified maximum, are distributed by randomly selecting two switches and connecting them. If either of both switches does not have a free port available, then the pair is discarded and a new pair is selected.

3.2.6 Real-world HPC Systems

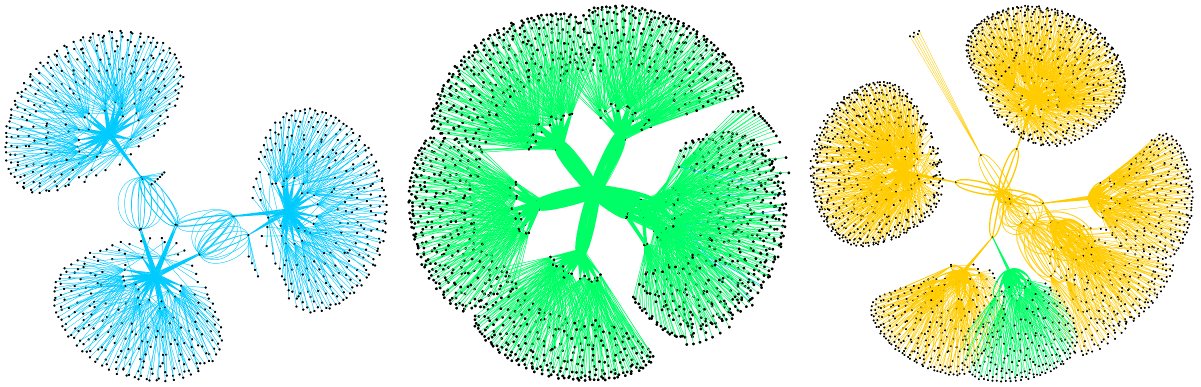
Alongside the theoretical network topologies depicted in the previous section, this thesis focuses on a number of real-world supercomputers and their interconnects:

- Deimos, formerly operated by the Technische Universität Dresden (TU Dresden), Germany,
- A successor of the Deimos HPC system, called Taurus, and
- The TSUBAME2.0/2.5 supercomputer operated by the Tokyo Institute of Technology, Japan.

These HPC systems will be analyzed with respect to their failure statistics, network/routing resiliency, and utilization during production. Subsequent network/routing suggestions and enhancement developed in the following two chapters are demonstrated on these systems.

The first HPC system, called Deimos [Juc08], was in operation from March 2007 to April 2012 at the Centre for Information Services and High Performance Computing, TU Dresden. The 726 compute nodes of Deimos hosted up to four AMD Opteron X85 dual-core CPUs and up to 32 GiByte of main memory, depending on the configuration. These compute nodes were connected to the InfiniBand fabric through a host channel adapter (HCA; i.e., InfiniBand's version of a network interface card) operating with single data rate (SDR) [Inf12]. The SDR IB fabric consisted of three 288-port director-class switches which have been connected in series, via 30 links per pair of director switches to be precise. Hence, the two outer director switches had 258 links to attach compute nodes, while the middle director switch only hosted 228 nodes. These switches themselves are internally constructed as 24-ary 2-trees using 24-port switch building blocks, resulting in a total of 108 IB switches and 1,653 IB links in Deimos. This three-island topology, see Figure 3.6a, was originally routed by the MinHop algorithm and towards the end of its lifetime routed by (DF-)SSSP, see Section 3.3.1 and Section 3.3.8, respectively.

In September 2010, the Tokyo Institute of Technology deployed the TSUBAME2.0 system [GSI12]. Its compute nodes are partitioned into three types, i.e., thin nodes, medium nodes, and fat nodes. Thin nodes host two hexa-core CPUs, as well as three GPUs, while the medium and fat nodes are designed as quad-socket plus single GPU with a main memory capacity of up to half a terabyte. All 1,408 thin nodes are connected to two separate InfiniBand fabrics through two QDR HCAs to facilitate the increased communication demand of the dense compute nodes. The first rail, or network plane, additionally



(a) Three-island SDR fabric of the Deimos HPC system **(b)** Full-bisection QDR fat-tree of TSUBAME2 (1st rail) **(c)** Multi-island QDR/FDR fabric of the Taurus supercomputer

Figure 3.6: Topology configurations of three real-world supercomputers; Colors indicate the utilized IB link speed: blue = SDR, green = QDR, yellow = FDR

includes connections to the medium and fat nodes, as well as supplementary I/O and administration nodes. Therefore, the total node count attached to the 1st rail is 1,555, as visualized in Figure 3.6b. Both fabrics, together consist of 501 36-port switches and 7,005 links, utilize QDR IB network components and assemble each a full-bisection bandwidth fat-tree topology. Originally, Up*/Down* routing has been used on both rail of TSUBAME2.0, which has been changed to fat-tree routing later, see Sections 3.3.4 and 3.3.5. In 2013, the petascale TSUBAME2.0 HPC system was upgraded to the newest GPU architecture and renamed to TSUBAME2.5, however the InfiniBand fabrics have not been changed. Therefore, the following chapters refer to this system as TSUBAME2.

The last multi-purpose petascale supercomputer listed above is a successor of Deimos at the TU Dresden, called Taurus [ZIH13]. Taurus' InfiniBand interconnection network is designed as a multi-island topology. The initial configuration, deployed in July 2013, consisted of three islands and a total of 494 compute nodes, and has been successively upgraded to the current size of six islands with 2,014 compute nodes plus 51 additional supplementary nodes. One 180-node island, depicted by the green color in Figure 3.6c, utilizes QDR IB technology, while the other islands use the FDR technology. Multiple smaller 2-level full-bisection fat-trees are connected by a 216-port director-class switch, as shown in Figure 3.6c, resulting in a total of 211 36-port switches and 4,580 links. The node configuration varies depending on the purpose of the island, e.g., HPC nodes are equipped with two 12-core Intel Haswell CPUs, single HCA, and up to 256 GiByte main memory. Throughput nodes utilize either Intel's Westmere or Sandybridge microarchitecture, and nodes with additional GPUs or I/O nodes are connected with two uplinks to the fabric, respectively. The entire Taurus fabric was routed with the fat-tree algorithm before switching to scheduling-aware routing, see Chapter 5, in August 2015.

Furthermore, additional failure analysis will be conducted in Chapter 4 for a cluster previously hosted by the Los Alamos National Laboratory. The supercomputer is referred to by "Cluster 2" [SG10] in the following. Unfortunately, very little configuration information has been published about this NUMA-based

Algorithm 3.1: SSSP: Global balancing of routes with single-source shortest-path routing

```
Input: Network $I = G(N,C)$, with node/switch set $N = S \cup T$ and $C$ as the set of links
Result: Linear forwarding table (LFT) for each switch
foreach switch $s_i \in S$ do
    foreach compute node $n_d \in T$, with $(n_d, s_i) \in C$ do
        Execute Dijkstra's algorithm with source node $n_d$ to calculate a spanning tree
        Update channel weights (add number of new paths to each used channel)
        Configure LFTs for all switches with $s_i$.LFT($n_d$)=port in reverse direction of the spanning tree
```

system, besides the system's failure data. While the deployed topology used to connect the 49 nodes is unknown, these 128-processor NUMA-nodes are attached to the network via 12 network interface cards each. The Cluster 2 at Los Alamos National Laboratory was in operation from April 1997 to August 2005.

3.3 Selection of Routing Algorithms for HPC

The following state-of-the-art routing algorithms will be used throughout the thesis for comparisons or will serve as the basis for further enhancements, as it is the case for the deadlock-free single-source shortest-path routing in Section 5.2. All these routing algorithms are implemented in the open-source subnet manager of InfiniBand (OpenSM) [Mel13]. The same holds for algorithms developed in this thesis, see Chapter 5 and 6.

3.3.1 MinHop Routing

MinHop routing [HSL09; Mel13] simply calculates the shortest path for each source/destination pair and assigns the liner forwarding tables (LFT) of the switches accordingly. Path balancing, for an optimized network throughput, is only performed on a local level, i.e., multiple links between two switches are balanced out. As long as the network is not bisected, MinHop will be able to find a shortest path for all source/destination pairs, and therefore is able to route arbitrary topologies. However, MinHop ignores the deadlock problem completely easily resulting in deadlocks on some topologies, which will be shown in Section 4.4.2.

3.3.2 Single-Source Shortest-Path Routing (SSSP)

SSSP routing, developed by Hoefler et al. [HSL09], is based on Dijkstra's algorithm which builds a spanning tree rooted at one node in the network. Dijkstra's algorithm is performed once for each node in the network to generate a path for each node pair. To balance the paths globally, the SSSP Algorithm 3.1 changes the edge weights of the graph, which represents the network, after the spanning tree for a node has been calculated. Compared to the MinHop and DOR routing algorithms (Section 3.3.3), SSSP is able to utilize higher network throughput due to the local and global path balancing. However, just as MinHop, SSSP routing neglects the routing-deadlock problem, and therefore has limited applicability for lossless interconnection networks.

3.3.3 Dimension Order Routing (DOR)

The dimension order routing [RR10, pp. 47-48], or XYZ-routing, is designed for mesh and hypercube topologies and is able to create a deadlock-free forwarding tables in the absence of failures. DOR, as it is implemented in the subnet manager, does not check the association of ports to dimensions and assumes a correct cabling, meaning that ports of the x-dimension must have a smaller port number then ports of the y-dimension. Therefore, it support every type of topology, but it can not guarantee deadlock-freedom for topologies other than meshes and hypercubes. Multiple links between two switches are balanced similar to MinHop.

3.3.4 Up*/Down* and Down*/Up* Routing

In the first step, Up*/Down* routing [Sch+91] tries to identify the root switches within the network. This is done via a statistical method, i.e., for each switch the algorithm calculates a distance to the HCAs, i.e., the network interface cards of InfiniBand, and correlates this to the number of shortest paths which would pass through this switch. An initial breadth-first search from the root switches is used to rank non-root switches depending on their distance to the root. Up*/Down* routing starts afterwards a breadth-first search (BFS) from each node in the system and updates the LFTs for all switches. While performing the BFS, the algorithm must not change the direction from "down" to "up" in order to avoid potential deadlocks. Limiting the number of links and directions potentially increases the congestion in the network. The Down*/Up* implementation of the OpenSM is similar to its counterpart, but it supports tree networks with HCAs connected to multiple levels of the tree.

3.3.5 Fat-Tree Routing

Zahavi et al. [Zah+10] introduced a routing algorithm, called fat-tree routing, which is designed for symmetrical or almost symmetrical k-ary n-trees and optimizes the LFTs for shift all-to-all communication patterns, see Section 4.3.3.2. Similar to Up*/Down*, deadlocks are prevented by placing restriction on links. The optimization for the shift pattern is a result of the algorithms ability to monitor the port usage for all downward ports at each switch. This static routing algorithm primarily consists of a main loop and two recursive functions, which are summarized in Algorithm 3.2 – 3.4, to evenly distribute the paths onto the available network links. Each switch is identified by a pair (h, i) depending on the level $0 \leq h < n$ in the tree and the index i within this level. Root switches are located at level 0. Algorithm 3.2 iterates over all leaf switches or attached compute nodes, respectively. The compute node in each iteration represents the destination for which the LFTs are updated. The tree-shaped network is then recursively traversed along the least loaded links by ascending one step towards the root level, and then descending down to all leaf switches which are part of the (sub-)tree, similar to a depth-first search, see Algorithm 3.3 and 3.4. Subsequently, the next step towards the root is performed. As well as Down*/Up*, fat-tree routing supports HCAs in each level of the tree. Providing a list of the root switches in the network is not necessary for Up*/Down*, Down*/Up*, and fat-tree. However, simulations conducted for Chapter 4

Algorithm 3.2: Fat-Tree: Setting LFTs towards all compute nodes

```
Input: Network $I = G(N,C)$, with node/switch set $N = S \cup T$ and links set $C$
Result: Linear forwarding table (LFT) for each switch
foreach leaf switch $s_i^h \in S$ do
    foreach compute node $n \in T$, with $(n, s_i^h) \in C$ do
        Obtain the LID of the compute node $n$
        Set $s_i^h$.LFT(LID)=port for the port connecting the compute node
        Call assign-down-going-port-by-ascending() for switch $s_i^h$
```

Algorithm 3.3: Fat-Tree: Assign-down-going-port-by-ascending

```
Input: Network $I = G(N,C)$, current switch $s_i^h$, destination LID
Result: Next switch $s_j^{h-1}$ towards root and modified LFT for $s_j^{h-1}$
/* Traverse down to leafs of (sub-)tree below $s_i^h$                    */
Call assign-up-going-port-by-descending() for $s_i^h$
if $h = 0$ then return
/* Traverse least loaded path towards a root switch                      */
Find the least loaded port $(s_j^{h-1}, s_i^h) \in C$ for all switches at level $h-1$
Assign $s_j^{h-1}$.LFT(LID) of the remote switch to that port
Increase port usage counter by one
Call assign-down-going-port-by-ascending() for $s_j^{h-1}$
```

Algorithm 3.4: Fat-Tree: Assign-up-going-port-by-descending

```
Input: Network $I = G(N,C)$, current switch $s_i^h$, destination LID
Result: Modified LFTs for all $s_\cdot^{h+1}$ attached to $s_i^h$
/* Recursively decent the (sub-)tree towards leafs                       */
foreach switch $s_j^{h+1} \in S$, with$(s_j^{h+1}, s_i^h) \in C$ do
    Find the least loaded port between $s_j^{h+1}$ and $s_i^h$
    Assign $s_j^{h+1}$.LFT(LID) to this port
    Increase port usage counter by one
    Call assign-up-going-port-by-descending() for $s_j^{h+1}$
```

Algorithm 3.5: LASH: Deadlock-free path calculation by layered shortest path routing

```
Input: Network $I = G(N,C)$, number of supported virtual layers $k$
Result: Linear forwarding table (LFT) for each switch and deadlock-free path-to-virtual layer assignment
Uses generalized version of DOR routing, see Section 3.3.3, to calculate shortest-paths for networks $I$
Combine virtual channels into virtual layers
Distribute all node pairs into multiple groups, each deadlock-free (e.g., all paths towards one $n_d$ in a group)
/* Assign groups to virtual layers without closing a cycle in the CDG */
while unassigned group exists do
    foreach $i = 1,\dots,k+1$ do
        Assign new group to CDG of virtual layer $L_i$
        if CDG of layer $L_i$ is cycle-free then
            break
        else
            Remove group from CDG and virtual layer $L_i$, respectively
    if $i == k+1$ then
        return false                                // no deadlock-free assignment possible
return true                                         // all potential deadlocks removed
```

showed that it improves the resulting network throughput and especially the resilience of the routing algorithms.

3.3.6 Layered Shortest Path Routing (LASH)

LASH and the following two routing algorithms are based on the method of splitting the physical link into multiple virtual channels. As described by Dally et al. [DS87], virtual channels can be used to avoid deadlocks by breaking the circular dependencies in the channel dependency graph. LASH was designed to generate a deadlock-free routing for arbitrary topologies [Lys+06]. The idea is to combine virtual channels into virtual layers and assign routes to the layers without closing a cycle in the corresponding channel dependency graph, as described in Section 3.1.2. LASH calculates the shortest paths between all switches, and tries to assign each path to the first virtual layer, as represented by Algorithm 3.5. If this would close a cycle in the CDG, then LASH checks the next virtual layer until it finds a suitable one. Since routes are calculated for switch pairs, all traffic between two switches can only utilize one single path in the network.

3.3.7 Torus-2QoS Routing

Torus-2QoS [Mel13] enhances the concept of DOR routing while being resilient to network faults and allowing deadlock-free routing on 2/3-dimensional meshes and tori networks. The dateline concept is used to assign paths to different virtual channels based on whether the route crosses a dateline in the torus or not. Torus-2QoS is able to reroute around a single link failure in each 1-dimensional ring of the torus by using the opposite direction, but not able to reroute around multiple link failures in the same ring. Switch failures are avoided by taking illegal dimension changes with regards to the DOR algorithm. Those illegal turns are routed via a different virtual channel to avoid deadlocks. An advantage of Torus-2QoS compared to LASH and DFSSSP is that a rerouting, due to link or switch failure, does not change the virtual lane assignment for a given HCA pair and therefore decreases the overhead for connection management in the

Algorithm 3.6: DFSSSP: Search cycles in the CDGs and remove potential deadlocks

```
Input: Network $I = G(N,C)$, linear forwarding tables, number of supported virtual layers $k$
Result: Deadlock-free path-to-virtual layer assignment
/* Initialize first channel dependency graph                                  */
foreach Path $P_{n_x,n_y}$ calculated by Algorithm 3.1 do
    Update the CDG of virtual layer $L_1$ with path $P_{n_x,n_y}$
/* Search for cycles in the channel dependency graphs                         */
foreach $i = 1,\dots,k-1$ do
    repeat
        Search for a cycle in the channel dependency graph of $L_i$
        if no cycle then
            break
        Identify weakest edge of the cycle
        foreach Path $P_{n_x,n_y}$ inducing this weakest edge do
            Remove path $P_{n_x,n_y}$ from the current virtual layer $L_i$
            Update CDG of virtual layer $L_{i+1}$ with path $P_{n_x,n_y}$
    until no cycle found in the CDG of $L_i$
Search for a cycle in the CDG of the last virtual layer $L_k$
if cycle found then
    return false                              // no deadlock-free assignment possible
Balance paths on empty CDGs without additional cycle search
return true                                   // all potential deadlocks removed
```

upper layers, e.g., in the Message Passing Interface (MPI) which is widely used across HPC applications.

3.3.8 Deadlock-Free SSSP Routing (DFSSSP)

Domke et al. [DHN11] combine the property of a globally balanced traffic of the SSSP routing with the deadlock-freedom using virtual layers, see Definition 3.8. The core of the SSSP algorithm, i.e., using Dijkstra's algorithm and updating edge weights, stays the same as described in Section 3.3.2. Since DFSSSP operates on HCA pairs instead of switch pairs, the amount of different paths is larger than for LASH, which makes the iterative approach of LASH impossible. Therefore, DFSSSP assigns all paths to the first layer and checks for cycles in the corresponding CDG afterwards. If a cycle is found, one edge in the cycle will be chosen and all paths, which induced this edge, are moved into the next virtual layer. The cycle search is repeated until the first layer is acyclic and the algorithm moves to the next layer. DFSSSP, as shown in Algorithm 3.6, iterates over all virtual layers until each layer is represented by an acyclic channel dependency graph.

4 Fail-in-Place High-Performance Networks

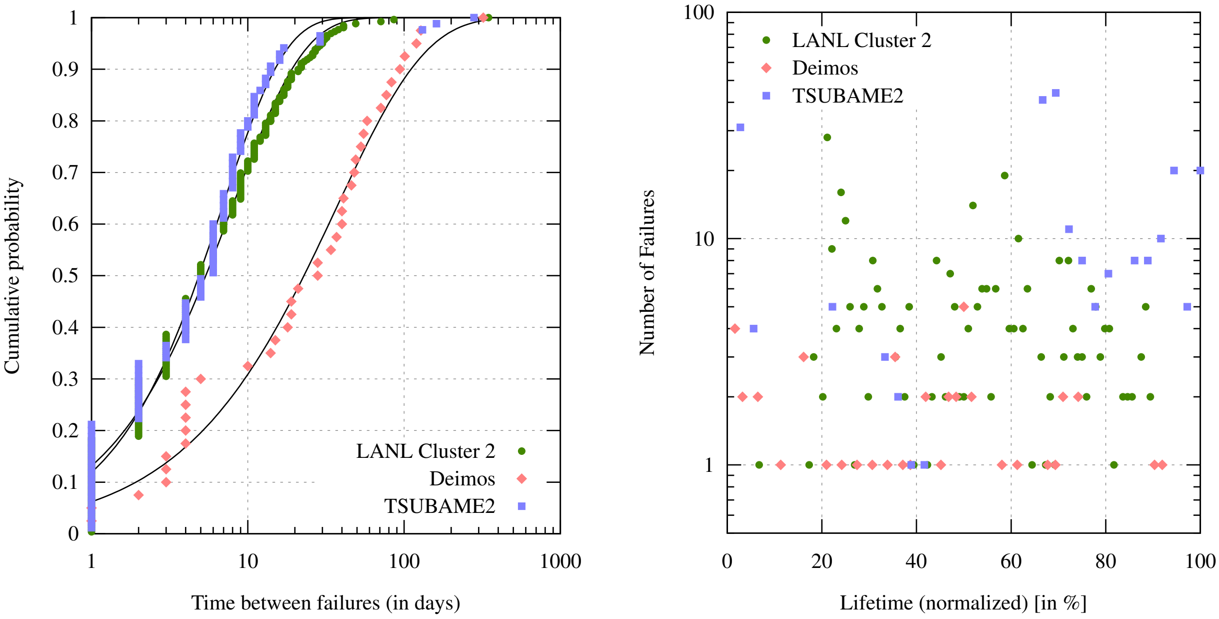
Similarly to other high-performance computing hardware, networks are not immune to failures. In this chapter the failure rates of three supercomputers will be analyzed, see Section 4.1. The subsequent Section 4.2 introduces the building blocks and metrics to construct and evaluate fail-in-place networks. Based on these analyses and based on a designed InfiniBand network simulation framework, see Section 4.3, a wide range of network simulations are performed. These simulations in Section 4.4 not only compare the throughput of different topologies, but also analyze the failure resiliency of different routing algorithms with their resulting throughput degradation. The results will be used to justify the consideration of cost-efficient fail-in-place HPC networks, assuming the right combination of topology and routing algorithm is deployed, instead of the state-of-the-art (deferred) maintenance model.

4.1 Failures Analysis for Real HPC Systems

The probability of network failures varies based on interconnect hardware and software, system size, usage of the system, and age. According to available fault statistics, network failures constitute between 2% – 10% for the HPC systems at Los Alamos National Laboratory [SG10], over 20% for local area networks (LANs) of Windows NT based computers [KKI99] and up to 40% for internet services [OGP03] of the total number of failures. Wilson et al. [Wil10] showed a fairly constant failure rate of $\approx$7 unstable links between switches per month and a total of $\approx$30 disabled network interface controllers (NIC) over the operation period of 18 month for a Blue Gene/P system.

The distribution of network component failures for three production systems, analyzed in this section, shows that network-related software errors are less common compared to other hardware errors, but the actual distribution of switch failures, NIC/HCA failures and link failures varies heavily, see Table 4.1. The first investigated HPC system, Deimos, in operation between March 2007 and April 2012 at TU Dresden, is a 728-node cluster, introduced in Section 3.2.6, with 108 IB switches and 1,653 links. The system's failure data is not publicly available, but was made accessible by the administrators of Deimos. The 49-node LANL Cluster 2 at Los Alamos National Laboratory was in operation from April 1997 to August 2005, and its failure statistics are available [Los14]. The same holds for the TSUBAME2 HPC system [Glo14]. Installed in September 2010, TSUBAME2 at Tokyo Institute of Technology uses a dual-rail QDR IB network with 501 switches and 7,005 links to connect its 1,408 compute nodes via two full-bisection bandwidth fat-trees. Further details about these systems are provided in Section 3.2.6.

Figure 4.1a shows the cumulative distribution function of hardware faults over the life span of all three



(a) CDF for faulty hardware

(b) Failures over time

Figure 4.1: Network-related hardware failures of three HPC systems

Table 4.1: Comparison of network-related hardware and software failures, MTBF/MTTR, and annual failure rates for three production supercomputers

Fault Type	Deimos	LANL Cluster 2	TSUBAME2[‡]
Percentages of network-related failures			
Software	13%	8%	1%
Hardware	87%	46%	99%
Unspecified		46%	
Percentages for hardware only			
NIC/HCA	59%	78%	1%
Link	27%	7%	93%
Switch	14%	15%	6%
Mean time between failure / mean time to repair			
NIC/HCA	X[†] / 10 min	10.2 d / 36 min	X / 5 – 72 h
Link	X / 24 – 48 h	97.2 d / 57.6 min	X / 5 – 72 h
Switch	X / 24 – 48 h	41.8 d / 77.2 min	X / 5 – 72 h
Annual failure rate			
NIC/HCA	1%	X	$\ll$ 1%
Link	0.2%	X	0.9%[*]
Switch	1.5%	X	1%

[†] Not enough data for accurate calculation.

[‡] Analysis limited to lifetime of TSUBAME2.0

[*] Excludes first month, i.e., failures sorted out during acceptance testing.

systems. Fitting the Weibull distribution to the data results in the following shape parameters for the different systems. For Deimos the shape parameter k is 0.76 and therefore the failure rate decreased over time [Nel04, pp. 36-39,143]. The shape parameters for TSUBAME2, which is $k = 1.07$, and for the LANL Cluster 2, $k = 0.95$, indicate a constant failure rate. While the constant failure rate applies to the LANL cluster, it does not hold for TSUBAME2. The hardware faults of TSUBAME2, see Figure 4.1b, show the expected bathtub curve for hardware reliability [VAK10, pp. 188-189].

Common practice in high-performance computing is to deactivate faulty hardware components and replace them during the next maintenance, unless the fault would threaten the integrity of the whole system, which may cause routing algorithms specifically designed for regular network topologies to fail. A failing switch can degrade the regular topology into an irregular topology which cannot be routed by a routing algorithm specifically designed for regular networks, even so the network is still physically connected. As shown in Table 4.1 the percentage of switch failures, among hardware-related failures of the network, ranges from 6% to 15% for the three systems. This covers issues of a faulty switch as well as faulty ports. One deactivated port of a switch will degrade a regular network topology, but replacing a whole switch for one broken port can be cost and time intensive.

For all network-related failures of the LANL Cluster 2 the calculated mean time between failure (MTBF) is 100.3 h and a mean time to repair (MTTR) is 56.2 min. If broken down into MTBF and MTTR for each network component, i.e., switch, link and network interface controller (NIC), then the numbers show that the MTTR is the smallest for NICs (36 min), followed by links (57.6 min) and switches (77.2 min). The MTBF for these components varies significantly. While, on average, every ten days one NIC experienced a failure, switches were four times as stable, i.e., their MTBF is equal to 41.8 d. The most reliable network components of LANL Cluster 2 were the links. Only 23 link failures within eight years of operation were recorded, which results in a mean time between failure of more than three months.

The system administrators of Deimos and TSUBAME2 followed a different approach of repairing faulty network components, so called partially deferred maintenance. While broken NICs (aka. HCAs in InfiniBand) of Deimos were replaced within an average of 10 min by spare nodes, repairing links or switches took between 24 h and 48 h. Faulty components of TSUBAME2 were disabled within minutes, but for a switch port failure only the port and not the entire switch was disabled. Those disabled parts were replaced within the range of 5 h to one business day, i.e., up to 72 h time to repair. An exception were links and ports between root switches. Those are replaced in the next maintenance period, which is scheduled twice a year.

Table 4.1 summarizes failure percentages, MTBF, MTTR, and annual failure rates of all systems. The failure data of TSUBAME2 indicates an annual failure rate of ≈1% for InfiniBand links as well as 1% for the used switches. Despite the previously mentioned varying failure rate for TSUBAME2's InfiniBand components, an assumed constant annual failure rate of 1% is a fair representative failure rate to simplify further analysis. In the later sections, extrapolations of the fail-in-place behavior of the network will be conducted. These extrapolations are based on an assumed system lifetime of eight years as well as based on the previously described annual failure rates.

4.2 Building Blocks for Fail-in-Place Networks

The fail-in-place property, in the context of interconnection networks, can be defined based on the differentiation between "critical" and "non-critical" network component failures. A critical component failure disconnects all paths between two hosts, whereas a non-critical failure only disconnects a subset of the paths between two hosts. The network fail-in-place strategy, as introduced in Chapter 1, is to repair critical failures only, but continue operation by bypassing non-critical failures. Hence, fail-in-place networks are only viable if the number of critical failures and resulting unavailability of the HPC system can be minimized, and additionally, the routing algorithm is able to circumvent non-critical failures. Therefore, the following section will identify the needed hardware and software characteristics for such a network.

4.2.1 Resilient Topologies

Theoretically, from the topology point of view, failures can be tolerated up to the point where a set of critical failures disconnects the network physically. Whether non-critical failures can disconnect the network through incomplete routing or not and their influence on the overall network throughput will be subject of the following sections. A high path redundancy in the topology enhances the fail-in-place characteristic of an interconnection network, as outlined in Section 3.1.3.

From the connectivity metric in Definition 3.9 it follows, that at least $\kappa(G)$ switches or $\lambda(G)$ links have to fail at the same time to create a critical, topology-disconnecting failure for $I = G(N,C)$. Depending on the failures rates of these components, see Table 4.1, the likelihood of this event is slim assuming a high base connectivity of the topology. Hence, some regular topologies are more suited than others with respect to their usage for fail-in-place networks, compare to Table 3.1. If switching between topologies is impossible or undesired, such as the renunciation of meshes for NoCs, then using redundancy $r_{C'}$ is an option to improve resiliency, essentially multiplying κ or λ by $r_{C'}$, respectively. The former can be achieved by duplicating the network and connecting the terminal nodes to each network plane, similar to the dual-rail approach followed by TSUBAME2, see Section 3.2.6. A less costly approach is the pure increase of the edge-connectivity by using a link redundancy larger than one, see Definition 3.3. However, this is only possible within the limits of the switch radix, e.g., current InfiniBand switches scale up to 36 ports, while Intel's Omni-Path switches are available with a maximum radix of 48.

4.2.2 Resilient Routing Algorithms

Improving the topology resiliency alone does not necessarily affect the fail-in-place characteristics of the network, i.e., when the routing algorithm cannot exploit the redundancy. Two classes of deterministic routing algorithms are suitable to build fail-in-place interconnects:

- Fault-tolerant topology-aware routings, and
- Topology-agnostic routing algorithms.

Theoretically, fault-tolerant topology-aware routings are able to compensate for minor faults within a regular topology, while the latter category is fault-tolerant by design due to their support for arbitrary/random topologies.

In terms of resiliency, topology-agnostic algorithms are superior to topology-aware routing algorithms when the network suffers from a high percentage of failures, as will be shown in Section 4.4.2. However, techniques to avoid deadlocks in topology-agnostic algorithms can limit their applicability for large networks, such as the 3-dimensional torus(7,7,7) studied in Section 4.4.3. Chapter 6 will take up this deadlock problem again and present a potential solution. The InfiniBand subnet manger, called OpenSM, currently implements nine deterministic routing algorithms, as listed in Section 3.3. Five of them, more precisely DOR routing, Up*/Down* and Down*/Up* routing, fat-tree routing, and Torus-2QoS routing, are representatives of the topology-aware algorithms. The remainder, namely MinHop routing, SSSP and DFSSSP routing, and LASH routing, are topology-agnostic routing algorithms. While DOR theoretically falls into the category of topology-aware routing algorithms, its implementation in OpenSM can be considered topology-agnostic but it may not be deadlock-free for arbitrary topologies [RR10].

Deadlock-freedom of the routing algorithm is, besides latency, throughput and fault-tolerance, an essential property to design fail-in-place networks. The simulations described in Section 4.4 reveal that even simple communication patterns such as all-to-all can cause deadlocks in the network. Once a deadlock is triggered the network either stalls globally or delays the execution if the interconnect possesses mechanisms for deadlock resolution.

Two other critical aspects of routing algorithms for fail-in-place networks are the runtime of the routing algorithm after a failure was discovered and the number of paths temporarily disconnected until the rerouting completed. Both problems depend on the routing algorithms and will be investigated hereafter.

4.2.3 Resiliency Metrics

To evaluate a fail-in-place network with its combination of topology and routing, common metrics such as bisection bandwidth cannot be applied directly, or analytical metrics such as the edge forwarding index might be too imprecise. Other factors, such as zero-load latency [DT03, p. 20], are less likely to be influenced by a non-critical failure since the high path redundancy of HPC networks usually prevents a fault-induced increase of the path length, and are therefore beyond the scope of this thesis. Important metrics are availability and throughput in view of the fact that non-critical failures occur, and these two factors will be analyzed in the following sections.

4.2.3.1 Disconnected Paths before Rerouting

The runtime of the routing algorithm to compute a new network state depends on various factors, such as topology, algorithm characteristics (e.g., topology-agnostic vs. topology-aware), and desired properties of the result, e.g., balanced, deadlock-free, shortest-path, etc. Often these properties may prevent an effective parallelization of the algorithm, as in case of deadlock-freedom for topology-agnostic routings which have to perform many acyclicity checks on large graphs [Bad99; Trä13]. The runtimes of state-of-the-art HPC routing algorithms ranges from milliseconds for small networks to minutes for larger systems with more than 4,096 terminals or HCAs, respectively [HSL09; DHN11; VSR15]. Until new LFTs are calculated a subset of the network might be disconnected. One important question is the number of disconnected paths after a link or switch has failed. Obviously, this number is constant and equal to $|N|-1$ for terminal-to-switch links. Each disconnected path potentially causes application crashes in an HPC system, if upper layer protocols fail to configure appropriate timeouts and/or retry counts.

First, the problem is analyzed analytically using the edge-forwarding indices $\gamma_R(c)$, introduced in Definition 3.12, for the links in I, i.e., the number of routes passing through link $c \in C$. To simplify matters, assume the routing function $R : C \times T \to C$ for a terminal graph $G(N,C,T)$, representing the interconnection network I, maps the current channel c_q of a path towards $n_d \in T$ to the next channel c_{q+1} for all switches $s \in S$. Let R^* be perfectly balanced, then the edge-forwarding index of I using this routing function is assumed to be minimal:

$$\gamma(I) := \gamma(I,R^*) = \min_R \gamma(I,R) = \min_R \max_{c \in C} \gamma_R(c) \tag{4.1}$$

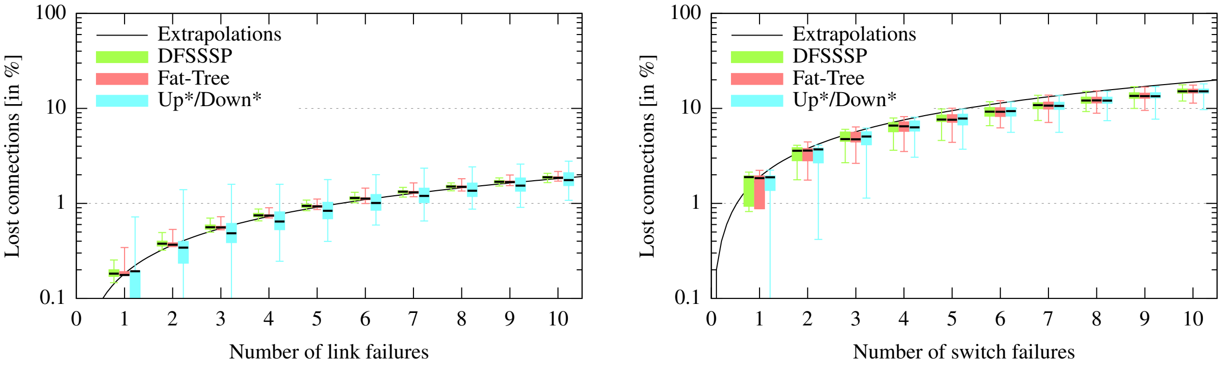
see [Chu+87; HS97] for further reference. Further, let $\gamma(c)$ be defined as $\gamma(c) := \gamma_{R^*}(c)$ for a fixed but arbitrary network I and perfectly balanced routing R^*. Removing a (faulty) channel c_f from I will disconnect a path P_{n_x,n_y}, predetermined by R^*, if and only if $c_f \in P_{n_x,n_y}$. A disconnected path will not contribute to any edge- or vertex-forwarding index in I. Obviously, the worst case scenario is loosing the link carrying the most paths, i.e., the link with $\gamma(I,R)$ for a given routing, and the best case is loosing c_f with $\gamma_R(c_f) = 0$. However, the latter case also denotes an imbalanced, and therefore inadequate, routing algorithm. Assuming the set of lost or disconnected links, before a new network state is calculated, is given by $C_L := \{c_f \mid c_f \in C \wedge c_{f_i} \neq c_{f_j} \text{ for } i \neq j\}$, then the average expected number of disconnected routes for a single failure $C_L = \{c_f\}$ can be calculated analytically:

Proposition 4.1 (Disconnected Paths by One Fault). *The expected number of disconnected routes* $\mathscr{E}(C_L)$ *due to one channel failure, given by the set* $C_L = \{c_f\}$, *is*

$$\mathscr{E}(\{c_f\}) = \frac{1}{|C|} \cdot \sum_{c \in C} \gamma(c)$$

with $0 < \mathscr{E}(C_L) \leq \gamma(I) \leq \gamma(I,R)$ *for any routing function R.*

Unfortunately, for $|C_L| > 1$ the number of disconnected paths $\mathscr{E}(C_L)$ has to be calculated as a conditional expected value, since some routes through c_{f_i} might have already been disconnected by removing another channel c_{f_j}. Hence, $\mathscr{E}(C_L)$ cannot be calculated exactly without knowing the exact graph, routing algorithm and set C_L. Thus, the following empirical analysis for a 10-ary 3-tree topology will show that this number can easily be approximated. For each number of link and switch faults, 100 topologies are created with randomly injected faults (seed $\in \{1,\dots,100\}$), and statistics of the number of lost



(a) Lost routes due to link failures

(b) Lost routes due to switch failures

Figure 4.2: Whisker plot for lost connections while removing links (4.2a) or switches (4.2b) from a 10-ary 3-tree without performing a rerouting; Topology contains 1,024 terminals and 2,000 links connecting the switches

connections are collected. To simplify matters, a simulated link failure consists of both directed channels within the link. The result, visualized by the whisker plots in Figure 4.2, shows that the measured extremes depend on the used routing algorithm, i.e., DFSSSP, fat-tree, and Up*/Down* routing. The whiskers indicate the lower and upper extremes, while each box shows the first/third quartiles and median across all 100 simulations for one specific number of faults. Figure 4.2b on the right demonstrate the percentage of disconnected paths when switches are removed, while the left Figure 4.2a show the same value for faulty switch-to-switch links. The two extrapolation curves in Figure 4.2 are based on the equation given in Proposition 4.2 showing the possibility to approximate $\mathscr{E}(C_L)$.

Proposition 4.2 (Disconnected Paths by Multiple Faults)**.** *For a small number of faults and assuming a large and arbitrary (but connected) network topology I, then the expected number of disconnected routes* $\mathscr{E}(C_L)$ *for* $|C_L| =: n > 1$ *can be modeled by the following approximation*

$$\mathscr{E}(C_L = \{c_{f_1}, \dots, c_{f_n}\}) \approx \frac{n}{|C|} \cdot \sum_{c \in C} \gamma(c)$$

Moreover, assuming a large number $|C_L| \gg 1$ of simultaneous failures, all arising before a new network state is calculated, indicates a more severe problem with the supercomputer requiring administrative actions, which is beyond the scope of fail-in-place network operation policies. Knowing the runtime of the routing algorithm and the estimated temporary disconnected paths due to a network failure allows to optimize predictions for system availability. Additionally, upper layer protocols can be tuned to set appropriate timeouts and retry counters, in order to mask the temporary disconnect.

4.2.3.2 Throughput Degradation Measurement for Reliability Analysis

Evaluating a network can be accomplished with multiple techniques and metrics. System designers often use analytical metrics, such as bisection bandwidth, which is simple to analyze for some regular topologies, but NP-complete in the general case, e.g., when failures deform the network. During system operation, metrics based on actual benchmarks are accurate in terms of reflection of the real world, but require exclusive system access to remove the background traffic in the network. Hence, the system's availability is reduced while performing these measurements. Therefore, a modeling technique is preferred, because it can be performed during the design phase of an HPC system and in parallel to the normal operation of the system, and parameters, such as topology, failure, routing, and communication pattern, can be modified.

Additionally, for the fail-in-place network design, one option is to rely on degradation modeling and the definition of a throughput threshold as a critical value. Modeling degradation over time in conjunction with a critical value, which defines the point of an unacceptable system state, are common in reliability analysis [Cro+14]. This thesis contributes a simulator to measure degradation and to highlight how communication throughput behaves for increasing numbers of failed network components. The simulator determines the throughput according to Definition 3.11 for two distinct network configuration, and the delta between the network throughputs quantifies the degradation. Section 4.4.2 introduces a statistical method based on the throughput degradation and linear regression to accurately compare the fail-in-place characteristics of different routing algorithms.

4.3 Interconnect Simulation Framework

Testing the hypothesis, whether or not fail-in-place networks could be a viable solution for current and future data centers and supercomputers, requires further analysis. Unfortunately, neither an analytical solution, nor the measurement approach are practical. The former fails due to the highly complex interaction between (faulty) topology, network technology, routing algorithm, and communication pattern, while the latter is too costly to construct a variety of large-scale networks, benchmarks these, and derive generally valid statements. Therefore, this thesis conducts wide-ranging network simulations as a more practical solution. The following sections illustrate how to construct this interconnect simulation framework.

4.3.1 Toolchain Overview

The first step in the simulation toolchain for network reliability modeling is the generation of the network, including network faults, see Figure 4.3. Either a regular topology, compare to Section 4.2.1, is created or an existing topology file is loaded, such as one of the production HPC systems from Section 3.2.6. The topology generator injects a number of user-defined and non-bisecting, i.e., non-critical, network component failures into the topology. These faulty components are uniform randomly selected using a seeded pseudo-random number generator to ensure repeatability. The user can also specify a fixed failure pattern reflecting an actual system state, e.g., a certain switch-to-link failure ratio or failures within a specific topology hierarchy level. Besides the actual network, the generator exports configuration files for the routing algorithms, such as the list of root switches for Up*/Down* routing.

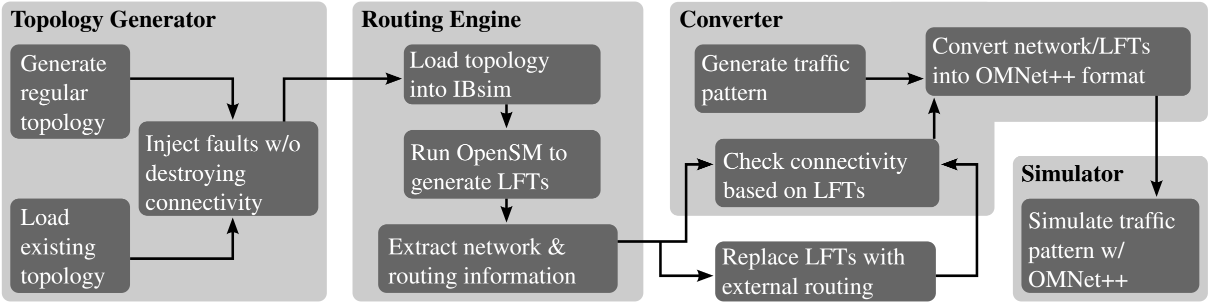

Topology Generator
Generate regular topology
Load existing topology
Inject faults w/o destroying connectivity
Routing Engine
Load topology into IBsim
Run OpenSM to generate LFTs
Extract network & routing information
Converter
Generate traffic pattern
Convert network/LFTs into OMNet++ format
Check connectivity based on LFTs
Replace LFTs with external routing
Simulator
Simulate traffic pattern w/ OMNet++

Figure 4.3: Toolchain to evaluate the network throughput of a fail-in-place network

The InfiniBand network simulator, called ibsim [Mel13], in conjunction with OpenSM is used to configure the linear forwarding tables (LFTs) of all switches with the selected routing algorithm. In an intermediate step, the LFTs can be replaced to simulate a routing algorithm not included in OpenSM, such as the routing algorithm for the 6-dimensional Tofu network of the K computer [Aji+12a]. The converter engine transforms the network and routing information into an OMNeT++ readable format.

A connectivity check is performed based on the calculated linear forwarding tables. Incomplete LFTs are considered a failed routing, even if the routing within OpenSM did not report any problems while generating the LFTs. The last step is the network throughput simulation with OMNeT++ [VH08] and the InfiniBand model [GR11].

OMNeT++ is one of the widely used discrete event simulation environments in the academic field. The InfiniBand model provides flit-level simulations of high detail and accuracy. The details of this existing IB model and configured parameters for subsequent simulations are briefly explained in Section 4.3.2. Furthermore, this thesis contributes additional extensions to the model, see Sections 4.3.3 and 4.3.4, which are required to execute the throughput degradation measurements for fail-in-place networks.

4.3.2 InfiniBand Model in OMNeT++

The simulated network itself consists of InfiniBand HCAs and switches and operates on the physical layer with an 8 bit/10 bit symbol encoding. Hence, this model theoretically supports SDR, DDR, QDR, as well as FDR-10 data rates [Inf12]. For later simulations, the network components are configured for 4X QDR (32 Gbit/s data transfer rate), while the HCAs are plugged into 8X PCIe 2.0 slots, offering a theoretical peak transfer rates of 5 GT/s and a throughput of 32 Gbit/s. The PCI bus has a simulated “hiccup” every 0.1 μs which lasts for 0.01 μs, both of which are configurable parameters of the IB model and which have been tuned to mirror the observed real world behavior on a small testbed, i.e., a slightly reduced consumption bandwidth at the destination nodes. This simulation environment further assumes that all links in the network are 7 m copper cables, which is the longest passive copper cable for 40 Gbit/s QDR offered by Mellanox. Therefore, the links cause a propagation delay of 43 ns for each flit, which is derived from the 6.1 ns/m delay mentioned in [BG07]. Switches are defined as a group of 36 ports connected via a crossbar which applies cut-through switching. Each switch input buffer can hold 128 flits per virtual lane, i.e., a total of 1,024 flits or 65,536 bytes, respectively. In contrast, switch output buffers can only store 78 flits. Messages are divided into chunks of message transfer units with a configured MTU size of 2,048 byte. Additionally, message header and footer, i.e., the local route header (8 byte), base transport header (12 byte), and in/-variant CRC (4 byte and 2 byte), are added which adds up to 2,074 byte for packets. The virtual lane arbitration unit within switches is configured for fair-share among all virtual lanes. All these parameters have been verified with a small network testbed, where it is possible to calculate the expected network throughput and consumption rate at the sinks. Furthermore, this testbed, consisting of 18 HCAs and one IB switch, is used to benchmark the InfiniBand fabric and PCI subsystem, and tune the simulation parameters accordingly.

4.3.3 Traffic Injection

The IB model uses unreliable connection (UC) for the transport mechanism to allow arbitrary message sizes and to omit the setup of queue pairs in the simulation. Hence, while using UC, the destination will not send an acknowledge message back to the source. However, since the model does not drop packets due to timeout or congestion, its throughput characteristics essentially reflects the more common reliable connection (RC) communication used in HPC interconnects. Two different traffic injection patterns, i.e., uniform random injection and an all-to-all exchange pattern, will be applied to estimate the throughput degradation of the networks, as introduced in Section 4.2.3. These patterns should show an application-agnostic approximation of the maximal achievable throughput for each combination of topology and routing.

4.3.3.1 Uniform Random Injection

The uniform random injection pattern is distinguished by the fact that all terminal nodes perform the following steps: (1) the terminal randomly selects the (next) destination, according to a uniform random distribution, (2) the terminal sends a message to this destination, and then goes back to step (1). This injection pattern should show the maximum throughput of the network which is measured at the sinks, i.e., the consumption bandwidth is extracted at each sink after the simulation reaches the steady state, see Section 4.3.4.1 for further details, or reaches a predefined wall-time limit. A seeded Mersenne twister algorithm [MN98], supplied by the OMNeT++ framework, provides the randomness of destinations and repeatability across similar network simulations where the only difference is the used routing algorithm. The message size for this type of traffic injection is 1 MTU for the simulations conducted in Section 4.4.2.

4.3.3.2 Exchange Pattern of varying Shift Distances

The all-to-all communication pattern used in various MPI-parallelized applications [Leó+16] is another pattern which strains the interconnect and required high network throughput. This pattern is distinguished by the fact that each terminal node sends a fixed amount of bytes to every other node in the network. One option to implement the all-to-all communication is an algorithm which uses the exchange pattern of varying shift distances. This algorithm determines the distances between all terminals for the network. Based on this distance matrix each terminal sends first to the closest neighbors with a shift of $\vec{s} = 1$ and $\vec{s} = -1$ and then in-/decrements the shift distance $\vec{s}$ up to $\pm\frac{|T|}{2}$ for the even case [Zah+10]. For an odd number of terminals attached to the topology, each terminal increments $\vec{s}$ up to $\pm\frac{|T|-1}{2}$ and subsequently performs the last shift of $\vec{s} = \frac{|T|+1}{2}$. This exchange pattern is similar to the commonly used linear exchange pattern [Pri+13a], except for the fact that it tries to optimize the traffic for full-duplex links. The assumption is that the reverse path uses the same links, and therefore incoming and outgoing messages between node pairs will not influence each others transfer rate. For the smaller simulation with approximately 256 terminals the message size is set to 10 MTUs and for larger networks reduced to 1 MTU to limit the simulation time. The network throughput for the exchange pattern is calculated based on a simplified version of Definition 3.11. Therefore, with the number of terminals, constant message size, and runtime, the throughput for the exchange pattern is given by the equation:

$$throughput := \frac{\#terminals \cdot (\#terminals - 1) \cdot message\ size}{runtime\ of\ exchange\ pattern} \tag{4.2}$$

4.3.4 Simulator Improvements

Simulations with the default IB model run either for a configured time or until the user terminates it, even so only flow control messages are send between ports. Therefore, neither does the simulator detect the completion of the simulated traffic pattern nor does it detect deadlocks if the routing algorithm is not able to create deadlock-free routing tables for the topology.

4.3.4.1 Steady State Simulation

In order to limit simulation time, the flit-level simulator detects the steady state of a simulation via a bandwidth controller module, which is used for traffic patterns with an infinite number of flits, such as explained in Section 4.3.3.1. If the average bandwidth (of all messages up to the “current” point in time) is within a 99% confidence interval, then the sink will send an event to the global bandwidth controller and report that the attached HCA is in a steady state. The global controller waits until at least 99% of all HCAs reported their steady state and then stops the simulation, assuming the network is in a steady state.

4.3.4.2 Send/Receive Controller

For InfiniBand simulations with finite traffic, such as the exchange pattern, the bandwidth controller is not applicable. Therefore, direct messages between the HCA generator/sink modules and a global send/receive controller are used. Each time the HCA creates a new message, it will determine the destination node, and send an event to the global controller. Sinks act similar and send an event each time the last flit of an IB message is received. The generators send a second type of event when they have created the last message. The global send/receive controller keeps track of all scheduled messages and stops the simulation when the last flit of a traffic pattern arrives at its destination.

4.3.4.3 Deadlock Controller

Deadlock detection is a complicated field and an accurate algorithm would require the tracking of actual flits and the available buffer spaces. Hence, to ensure relatively fast executions, the IB simulator uses a low-overhead distributed deadlock detection which is similar to the hierarchical deadlock detection protocol proposed by Ho and Ramamoorthy [HR82]. A local deadlock controller will observe the state of each port within a switch and reports periodically the state to a global deadlock controller. The three states of the switch are: idle, sending, and blocked (i.e., flits are waiting in the input buffers, but there is no free buffer space in the designated output port). The global deadlock controller will stop the simulation if and only if the state for each switch in the network is either blocked or idle.

4.4 Simulation Results

The previously introduced framework is used in this section to analyze and compare the fail-in-place properties of different topology and routing combinations for the InfiniBand architecture. The initial study determines whether or not a routing algorithm is applicable to a certain type of topologies. Thereafter, multiple smaller and large-scale topologies are investigated with respect to their throughput degradation given an injected amount of network link failures, see Sections 4.4.2 and 4.4.3. Finally, two of the HPC systems introduced in Section 3.2.6 are examined regarding their fault resiliency for link or switch failures, or a combination of both.

4.4.1 Initial Usability Study

Not all routing algorithms work for all topologies. The first part of this study characterizes each routing algorithm for each topology regarding the following two questions:

- Is the algorithm able to generate valid routes for the topology?
- Are these routes deadlock-free?

Table 4.2 lists all used topologies, i.e., the failure-free configuration, for this test.

The topology generator of the simulation framework constructs the input graph for ibsim. Afterwards, OpenSM routes the network. If the routing algorithm creates a valid set of routes, then the network and forwarding tables are transformed into the input format for OMNeT++. Otherwise, the combination of topology and routing is marked with a red box in Table 4.3. OMNeT++ simulates the uniform random injection workload on the interconnect and checks for deadlock situations. The result can be seen in Table 4.3. Furthermore, results for Down*/Up* routing are omitted in the following since all executed simulations have not shown any difference in performance compared to Up*/Down* routing.

As expected, all topology-aware routing algorithms are able to route their specific topologies, but usually not other types. Note that dimension order routing (DOR) did not produce a deadlock on the 3-dimensional torus(3,3,3) while the 5x5 2D torus configuration deadlocked. This is due to the one-hop

Table 4.2: Topology configurations for a balanced (and unbalanced) number of HCAs with link redundancy $r_{C'}$ to allow a fair comparison between topology types

Topology	Switches	HCAs	Links	$r_{C'}$
2D mesh(5,5)	25	275 (256)	240	6
3D mesh(3,3,3)	27	270 (256)	216	4
2D torus(5,5)	25	275 (256)	300	6
3D torus(3,3,3)	27	270 (256)	324	4
Kautz(2,4)	24	264 (256)	288	6
16-ary 2-tree	32	256 (270)	256	1
XGFT(1,22,11)	33	264 (256)	242	1
Dragonfly(10,5,5,4)	40	280 (256)	276	1
Random	32	256 (270)	256	1

Table 4.3: Usability of topology and routing combinations for networks listed in Table 4.2; Colors indicates: o = deadlock-free; r = routing failed; d = deadlock detected

	Fat-tree	Up*/Down*	DOR	Torus-2QoS	MinHop	SSSP	DFSSSP	LASH
artificial topologies								
2D mesh	r	r	o	o	d	d	o	o
3D mesh	r	r	o	o	d	d	o	o
2D torus	r	r	d	o	d	d	o	o
3D torus	r	r	o	o	d	d	o	o
Kautz	r	r	d	r	d	d	o	o
k-ary n-tree	o	o	o	r	o	o	o	o
XGFT	o	o	o	r	o	o	o	o
Dragonfly	r	r	d	r	d	d	o	o
Random	r	r	o	r	d	d	o	o
real-world HPC systems								
Deimos	r	o	o	r	o	o	o	o
TSUBAME2	o	o	o	r	o	o	o	o
	topology-aware				topology-agnostic			

paths on the 3x3x3 torus. However, DOR will produce deadlocks for larger 3-dimensional tori because it is generally only deadlock-free on meshes.

In contrast, all topology-agnostic algorithms were able to route all topologies. However, MinHop and SSSP do not prevent deadlocks and thus create deadlocking routes in some configurations. Hereafter, only topology/routing pairs are analyzed which are marked as deadlock-free in Table 4.3, with the exception of the DOR routing algorithm on 3-dimensional tori.

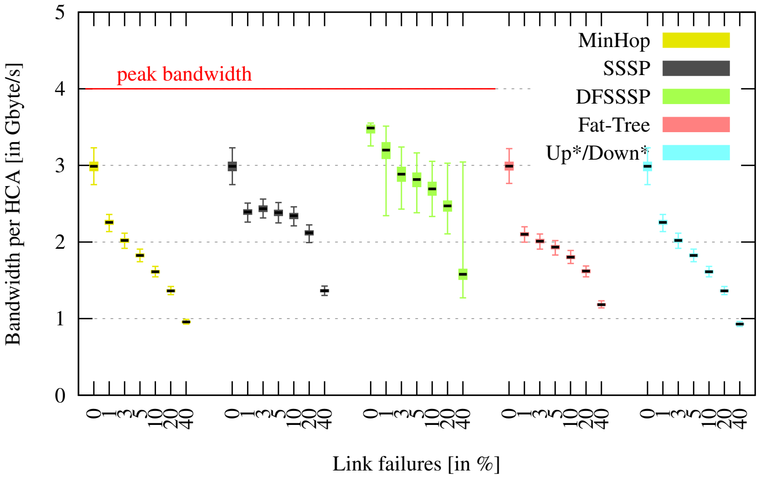
4.4.2 Influence of Link Faults on Small Topologies

In this section, OMNeT++ and the IB model simulate uniform random injection and exchange communication patterns for all correct and deadlock-free routing algorithms. The determined consumption bandwidths at the HCAs compose the simulation results while using the uniform random injection pattern. The maximum injection and consumption bandwidth is 4 Gbyte/s, because 4X QDR is simulated. In contrast, for the exchange pattern, the time to complete the full communication is determined by measuring the elapsed time until the last flit arrives at its destination. The resulting overall network throughput for this pattern is calculated according to Equation 4.2.

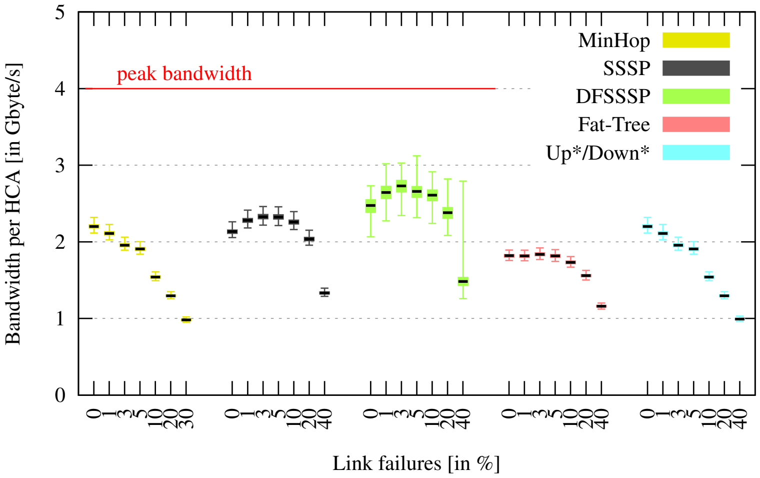
The simulated failure scenarios are: 1%, 3%, and 5% injected random link failures, with three different seeds for uniform random injection and ten seeds for the exchange pattern. Flich et al. [Fli+12] performed similar studies with these failure percentages. In addition, more severe failure scenarios are simulated with 10%, 20%, and 40% random link failures, as outlined in Section 4.1, to unveil long-term effects of a fail-in-place environment and to unveil deficiencies of individual routing algorithms beyond the results obtained in the Section 4.4.1. Whisker plots are used to show the results for uniform random injection to illustrate the consumption bandwidth distribution among all HCAs in the network. Furthermore, the presented values are averaged across three topology configurations, given by the same failure percentage but using different seeds for the fault injection. In contrast to this, histograms with error bars are used to show the network throughput of exchange communication patterns, where ten different seeds are utilized to obtain varying topologies for the same failure percentage. The mismatch in applied seeds is due to a higher required runtime for the simulator to reach a steady state for the uniform random injection pattern.

Two configurations, a balanced and an unbalanced, in terms of the number of HCAs per switch are simulated to see whether this has an influence on the maximal network throughput and resiliency or not, see Table 4.2. The number of HCAs for the unbalanced configuration is listed in brackets in the third column of Table 4.2. Figure 4.4a and Figure 4.4b show the balanced vs. unbalanced comparison while omitting results of DOR and LASH routing, because their resulting consumption bandwidth is less than 10% of DFSSSP's.

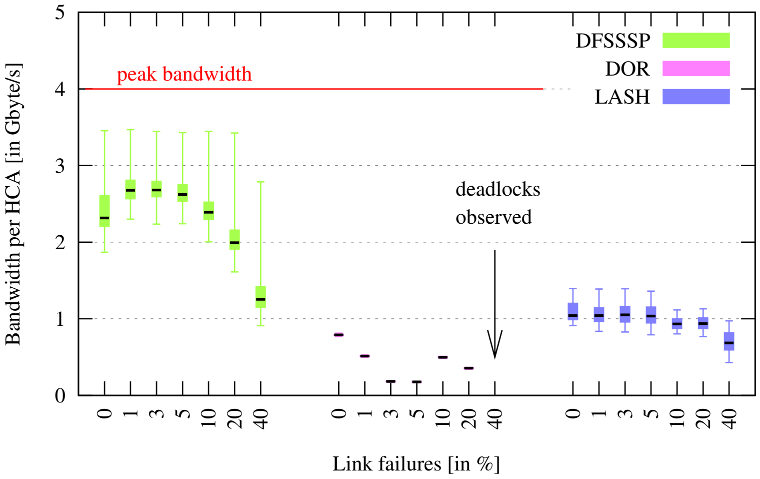
Analyzing the edge forwarding index for each simulation revealed that LASH heavily oversubscribes certain links in the network while other links are not used at all. Therefore, heavy congestion leads to the aforementioned consumption bandwidth decrease, regardless of the link failure rate. A possible explanation is that the implementation of LASH in OpenSM optimizes for switch-to-switch traffic, see



(a) Balanced 16-ary 2-tree with 256 HCAs



(b) Unbalanced 16-ary 2-tree with 270 HCAs



(c) Random topology with 32 switches, 256 HCAs

Figure 4.4: Whisker plots of consumption bandwidth for uniform random injection (box represents the three quartiles of the data; end of the whisker shows the minimum and maximum of the data; x-axis is not equidistant); Shown are the average values for three seeds (seed $\in \{1,\ldots,3\}$); Discussion in Section 4.4.2

Section 3.3.6, rather than terminal-to-terminal traffic. This leads to poor path balancing, and therefore poor load balancing, on many of the investigated topologies.

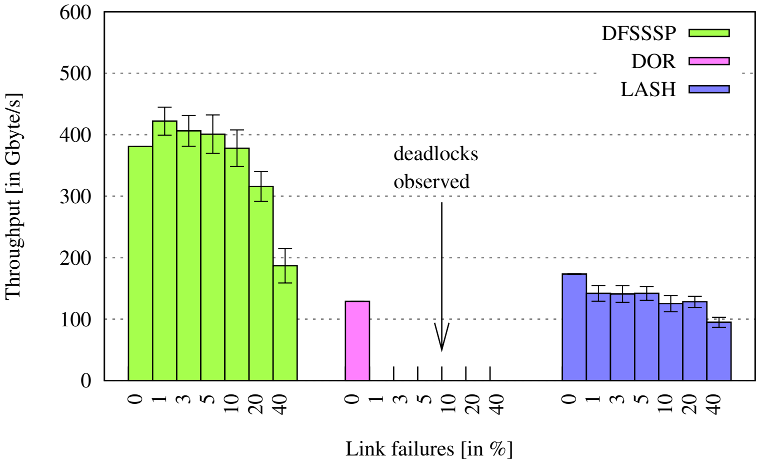
A comparison of Figure 4.4a and Figure 4.4b reveals three general trends: First, only 1% link failures, i.e., two faulty links for the 16-ary 2-tree topology, may result in a throughput degradation of 30% for uniform random injection on a fat-tree with full bisection bandwidth if the specialized fat-tree routing is used. With DFSSSP, for example, the degradation is only 8% for the median. Thus, the conclusion is that the choice of the routing method is important to maintain high bandwidths in the presence of network failures. Second, an imbalance in the number of HCAs per switch can have a similar influence on the throughput, even without faults in the network ($\approx$40% degradation compared to the balanced fault-free case). Hence, while the resiliency of all routing algorithms is not affected by an unequal number of HCAs, which can be caused by compute node failures or caused by initial network design decisions, the network throughput will be affected. Third, a curious case can be observed where the bandwidth increases with link failures, especially for DF-/SSSP in Figure 4.4b. This indicates a suboptimal routing algorithm and the effect will be explained in detail later.

Figure 4.4c and Figure 4.5a show the two communication patterns uniform random injection (Figure 4.4c) and exchange (Figure 4.5a) for the same random topology. A finding is that random topologies are barely affected by link failures of less than 10%. For the following analyses in Sections 4.4.3 – 4.4.5, the results for the uniform random injection are omitted. The reason is that an observed low consumption bandwidth at the sinks perfectly correlates with a runtime increase of the exchange pattern, and therefore decrease in overall network throughput, and vice versa.

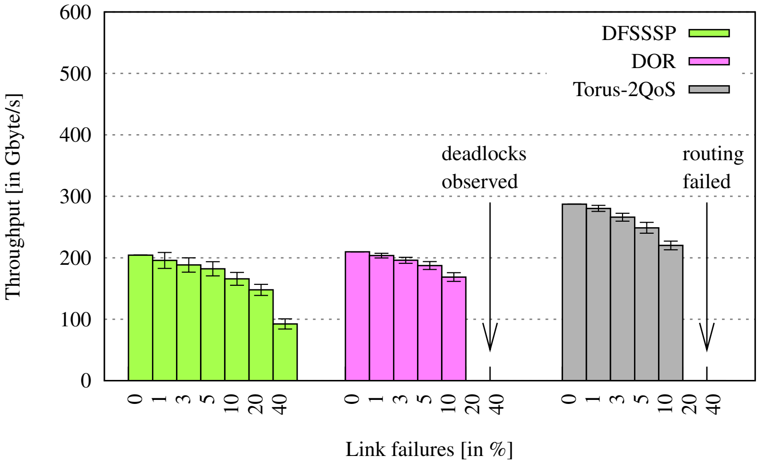
Torus performance is shown in Figure 4.5b. The exchange pattern finishes 29% faster, i.e., 29% higher throughput, for Torus-2QoS routing compared to DFSSSP. However, DFSSSP is more resilient, because Torus-2QoS was unable to create LFTs for some of the ten cases of 20% and 40% link failures. Similarly problematic, the simulated traffic pattern caused a deadlock using DOR routing for the mesh with 20% and 40% link failures. Again, LASH is omitted due to very low throughput for this topology.

Dragonfly topologies are usually routed with the Universal Globally-Adaptive Load-balanced routing scheme [Kim+08], which chooses between minimal path routing and Valiant's randomized non-minimal routing depending on the network load. The current architecture specification of InfiniBand mandates deterministic routing [Inf15, p. 234], but for completeness this study includes simulations for the emerging dragonfly topology as well. The reason behind this is that a decreased congestion on the minimal path through a well-balanced routing, such as DFSSSP, potentially reduces the number of adaptive routing decision via non-minimal paths in a dragonfly network. Figure 4.5c shows that both DFSSSP and LASH only degrade slowly with random network failures.

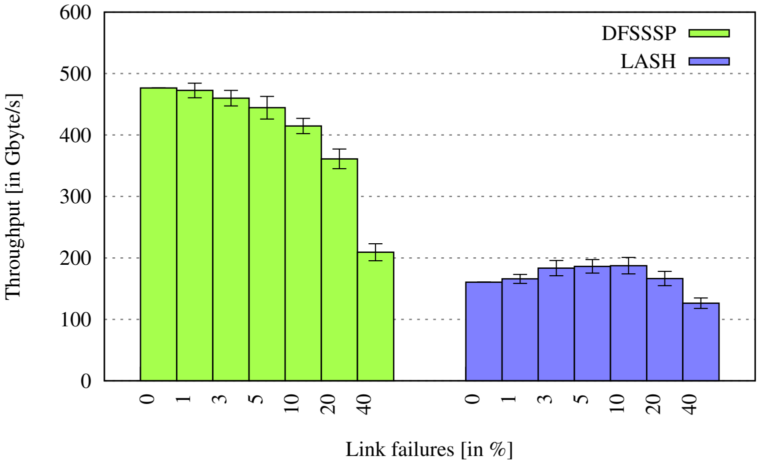
Simulating multiple failure scenarios with different seeds allows to derive conclusions through statistical methods, as outlined hereafter. Hence, one option is the analysis of the coefficient of determination (R^2), assuming a linear model, for every combination of topology and routing algorithm. This statistical analysis of the data points for the exchange pattern for all investigated topologies shows a high correlation between the number of failed links in the topology and the network throughput of the exchange pattern.



(a) Random topology with 32 switches, 256 HCAs



(b) Balanced 3D mesh$(3,3,3)$ with 270 HCAs, $r_{C'} = 4$



(c) Balanced dragonfly$(10,5,5,4)$ with 280 HCAs

Figure 4.5: Histograms for exchange patterns with error bars (showing mean value and the 95% confidence interval for all ten seeds (seed $\in \{1,\ldots,10\}$)); Missing bars: Deadlocks observed using DOR routing for at least one out of ten seeds, Torus-2QoS routing failed (multiple link failure in one 1D ring); Discussion in Section 4.4.2

Table 4.4: Intercept [in Gbyte/s], slope, and R^2 for balanced topologies of Table 4.2 for best performing routing algorithm

Topology	Routing	Intercept	Slope	R^2
2D mesh	Torus-2QoS	263.63	-1.83	0.94
3D mesh	Torus-2QoS	276.18	-2.37	0.87
2D torus	Torus-2QoS	341.11	-1.68	0.95
3D torus	DFSSSP	508.97	-2.12	0.90
Kautz	DFSSSP	299.48	-1.45	0.88
16-ary 2-tree	DFSSSP	629.76	-3.59	0.69
XGFT	DFSSSP	527.31	-3.03	0.88
Dragonfly	DFSSSP	479.03	-2.39	0.94
Random	DFSSSP	417.53	-2.17	0.76

Thus, this thesis contributes an approach for determining the fail-in-place characteristics, as part of the design process for a new fail-in-place network, based on the following methodology: The best routing algorithm, among the investigated algorithms, for a fail-in-place network will be chosen based on the intercept and slope of the linear regression. The intercept approximates the overall network throughput of the exchange pattern for the fault-free network configuration whereas the slope predicts the throughput decrease (in Gbyte/s) for each non-critical hardware failure which is kept in-place. Therefore, even hybrid approaches are possible where the system administrator switches between routing algorithms after a certain number of failures, if the regression line of the two routing algorithms intercept at this point.

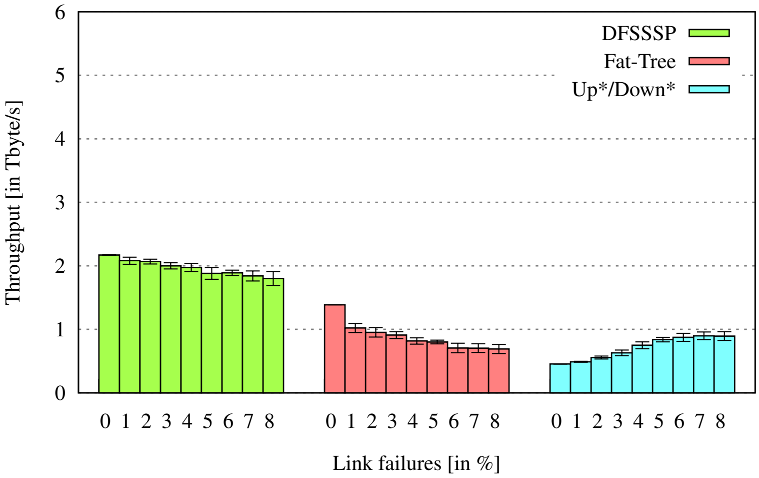
Intercept and slope are the main indicators for the quality of the routing whereas the coefficient of determination is an important metric how well the throughput can be explained by the number of failures. The results for the coefficient of determination calculation for previously investigated topology/routing combinations are summarized in Table 4.4. This shows that the performance of the exchange pattern on small topologies can be explained almost entirely by one variable, namely link failures. Table 4.4 also lists the best performing routing algorithm based on intercept and slope for each topology.

4.4.3 Influence of Link Faults on Large Topologies

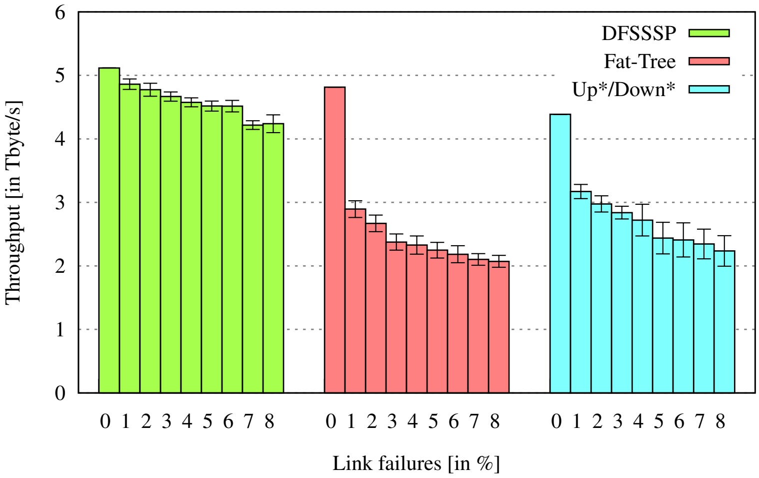
For the larger network sizes, the percentage of link failures is reduced to more realistic values found in modern HPC systems. A constant annual failure rate of 1%, derived from the analyzed failure rates of TSUBAME2 in Section 4.1, is a reasonable assumption. Hence, simulating $1\%,\ldots,8\%$ links failures will give an estimate of the performance degradation of the network for the system's lifetime.

Investigating larger topologies reveals three key points: First, the investigated fault-tolerant topology-aware routing algorithms are resilient for realistic link failure percentages, but their fail-in-place characteristic in terms of network throughput is worse than topology-agnostic algorithms. The comparison of DFSSSP routing and fat-tree routing in Figure 4.6b illustrates the issue.

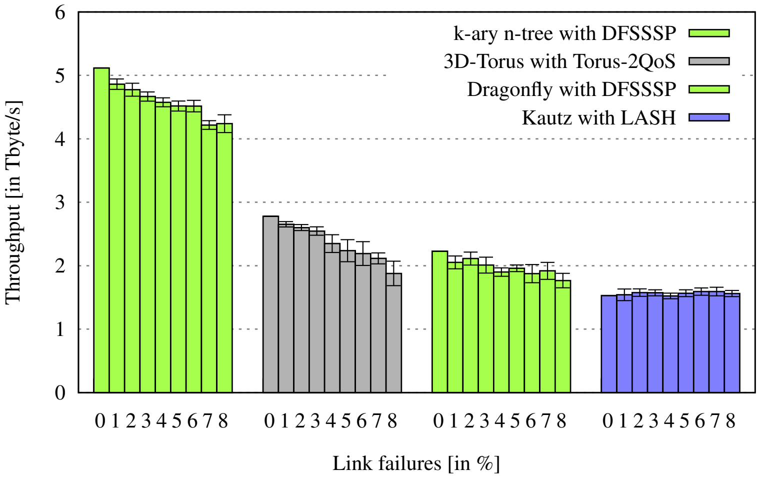
The second fact is that not all topology-agnostic routing algorithms are applicable to every topology



(a) 10-ary 3-tree with 1,100 HCAs



(b) 14-ary 3-tree with 2,156 HCAs



(c) Different networks with ≈2,200 HCAs

Figure 4.6: Histograms (mean value and 95% confidence interval; ten runs with different seeds $\in \{1, \dots, 10\}$) for exchange patterns for networks with up to 8% link failures; Figure 4.6c compares 14-ary 3-tree with 2,156 HCAs, 3D torus(7,7,7) with 2,058 HCAs, dragonfly(14,7,7,23) with 2,254 HCAs, and Kautz(7,3) with 2,352 HCAs (from left to right) for the best performing routing; Discussion in Section 4.4.3

Table 4.5: Intercept [in Gbyte/s], slope, and R^2 for the large topology/routing combination shown in Figure 4.6c

Topology	#HCAs	Routing	Intercept	Slope	R^2
3D torus(7,7,7)	2,058	Torus-2QoS	2,794.89	-2.06	0.66
Dragonfly(14,7,7,23)	2,254	DFSSSP	2,166.54	-1.50	0.34
Kautz(7,3)	2,352	LASH	1,541.58	0.09	0.02
14-ary 3-tree	2,156	DFSSSP	5,015.53	-1.85	0.76

above a certain size. Figure 4.6c shows the comparison of throughput degradation between four balanced large-scale topologies with ≈2,200 InfiniBand HCAs for the best performing (or solely applicable) routing. DFSSSP outperforms the rest for the dragonfly topology and the 14-ary 3-tree, but fails to create LFTs for the torus and the Kautz graph due to limitations of available virtual lanes. The same issue causes LASH routing to be unusable on the 7x7x7 torus, rendering Torus-2QoS the only applicable algorithm.

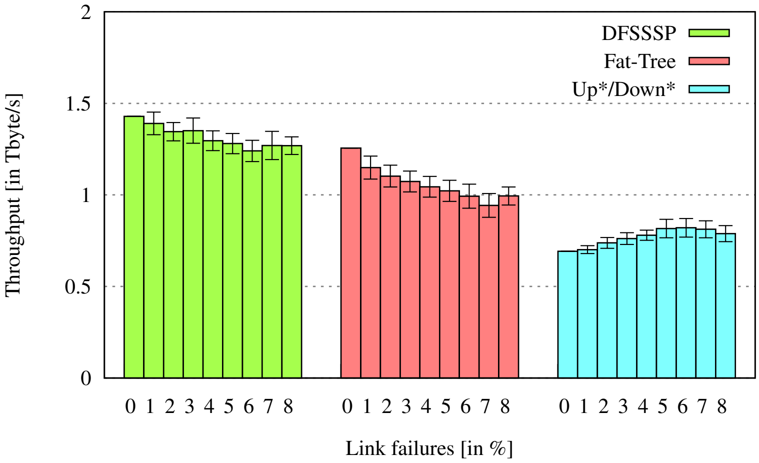
And third, the low coefficient of determination R^2, see Table 4.5, indicates that the number of link failures is not the dominating influence on the runtime of the exchange pattern on larger topologies. Hence, a low R^2 for the best of the investigated routing algorithms means that the throughput does not decline heavily during the system's lifetime which is a desired characteristic for a fail-in-place network. Presumably, the match or mismatch between the configured static routes and the communication pattern is a second explanatory variable, which will be explained further based on Figure 4.6a.

Studying Figure 4.6a exposes an anomaly: A major increase in network throughput using Up*/Down* routing is possible while the number of failed links in the network increases. This contradictory effect has two reasons. First, a comparison with DFSSSP and fat-tree routing, which enable higher throughput, indicates a serious mismatch between communication pattern and static routing algorithm in the fault-free case, which has been shown by Hoefler et al. [HSL08]. The second reason is that each failure in the fabric will change the deterministic routing, which can lead to an improvement in runtime for the same communication pattern. This change in the routing can remove congestion, because simultaneously executed point-to-point communications share less links/switches in the fabric.

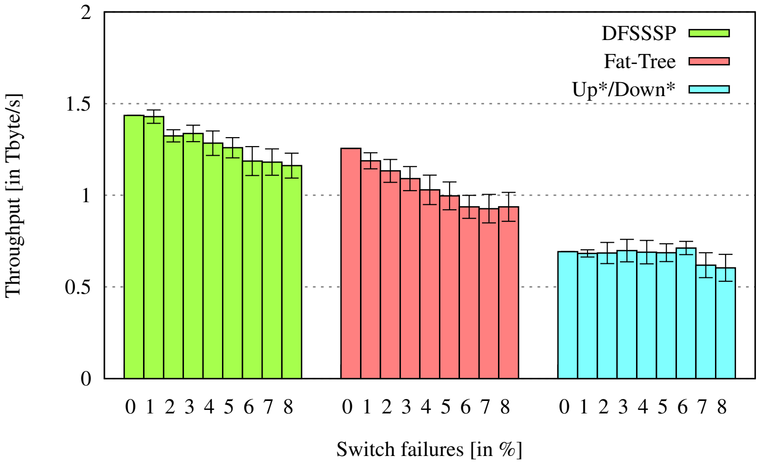
4.4.4 Case Study of the TSUBAME2 Supercomuter

Randomized link failures are combined with switch failures for the TSUBAME2 case studies in this section. These simulations with additional switch failures, i.e., a form of localized simultaneous failures of many links, mirror the behavior of a real-world HPC system more accurately.

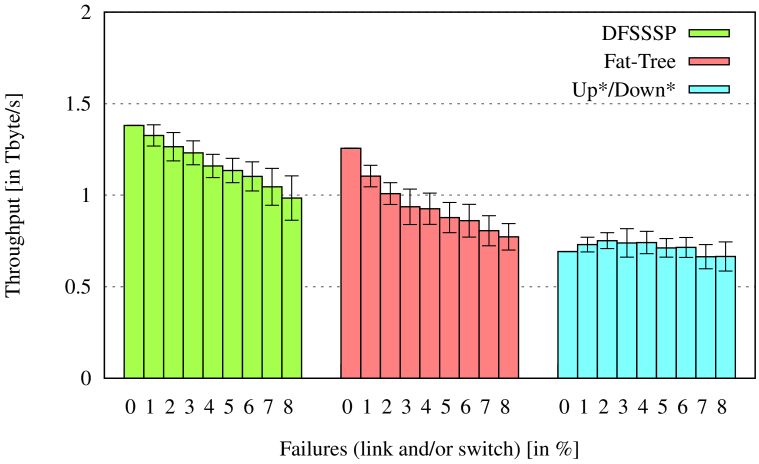
Based on the fault data presented in Section 4.1, the calculated annual link failure rate is 0.9% and the annual switch failure rate is 1% for TSUBAME2. The same data indicates a failure ratio of 1 : 13, i.e., thirteen switch-to-switch links will fail for each failing switch on average per year. In the following, a constant annual failure rate of 1% is assumed for the links and switches. Furthermore, the simulations will be conducted based on the 1st rail, i.e., including all 1,555 nodes which are connected by 258 QDR



(a) Link failures only (1% annual failure rate)



(b) Switch failures only (1% annual failure rate)



(c) Switch and link failures (1 : 13 ratio)

Figure 4.7: Histograms for exchange patterns for different failure types using DFSSSP, fat-tree and Up*/Down* routing on TSUBAME2 (LASH excluded because the throughput is only 2% of DFSSSP's throughput); Shown: mean value and 95% confidence interval for ten runs per failure percentage (seeds $\in \{1, \dots, 10\}$); Discussion in Section 4.4.4

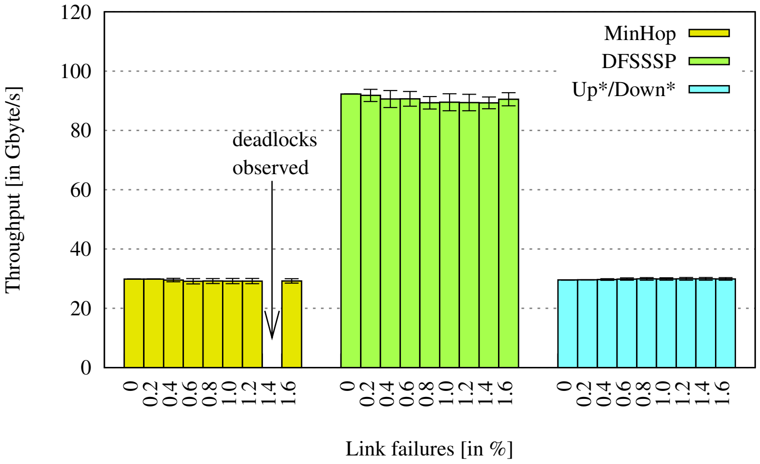
InfiniBand switches and 3,621 links. Three different failure cases, all shown in Figure 4.7, are simulated by the developed toolchain with ten different seeds per failure percentage. Figure 4.7a shows the simulated performance degradation for the case when only links fail within the supercomputer. Furthermore, the second case of solely failing switches is visualized in Figure 4.7b. In addition, the failure ratio of 1 : 13 is investigated in Figure 4.7c to mirror the real-world behavior and to analyze the network performance and resiliency of the routing algorithms for TSUBAME2. Going up to a 8% failure percentage results in the simulation of eight years of TSUBAME2's lifetime. Currently, the anticipated lifetime of the network used in TSUBAME2 is seven years, hence these simulations cover the entire lifespan of the system. The results for Down*/Up* routing are excluded in all figures, since they are equal to Up*/Down* routing. Similarly, results for LASH routing are omitted, because LASH only allowed for a two orders of magnitude lower throughput, see Table 4.6. The default routing algorithm on TSUBAME2 was Up*/Down* before switching to fat-tree routing. A reasonable modification to the operation policies of TSUBAME2, since fat-tree routing performs better for an exchange pattern, as can be seen in Figure 4.7a. Nonetheless, the simulated performance degradation also suggests that DFSSSP routing outperforms the other two algorithms.

Statistical analysis of the data presented in Figure 4.7c shows that the intercept for fat-tree routing is 15% lower compared to DFSSSP routing while the slope of the linear regression line is 10% steeper compared to DFSSSP, see Table 4.6. In conclusion, the change of the routing algorithm to DFSSSP on TSUBAME2 would not only lead to a higher performance of the exchange pattern on the fault-free network, but will also increase the fail-in-place characteristic of the network. The DFSSSP routing algorithm will provide a basis for the novel routing approaches outlined in Chapter 5 and Chapter 6 of this thesis.

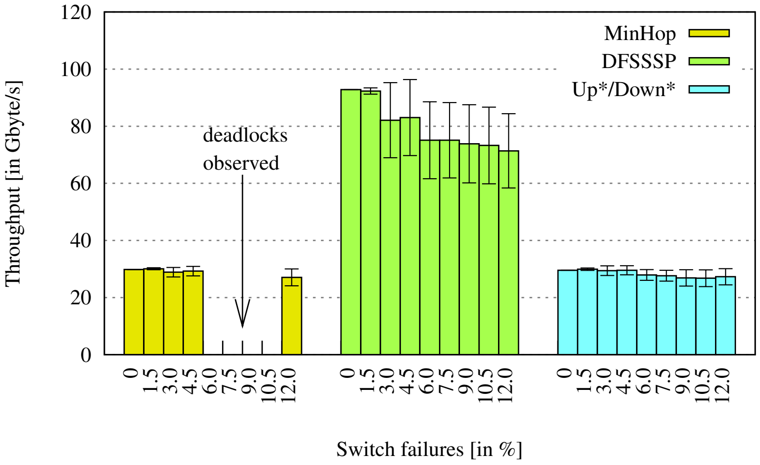
4.4.5 Case Study of the Deimos HPC System

Similarly to the previous section, link failures and switch failures are used individually and in a combined configuration to simulate the fail-in-place characteristics of the Deimos HPC System. The annual failure rate is 0.2% for links and 1.5% for switches, as listed in Table 4.1. Hence, the switch-to-link fault ratio can be approximated with 1 : 2. For each number of link and/or switch failures, ten different faulty network configuration are simulated, while injecting randomized failures with seeds $\in \{1, \dots, 10\}$.

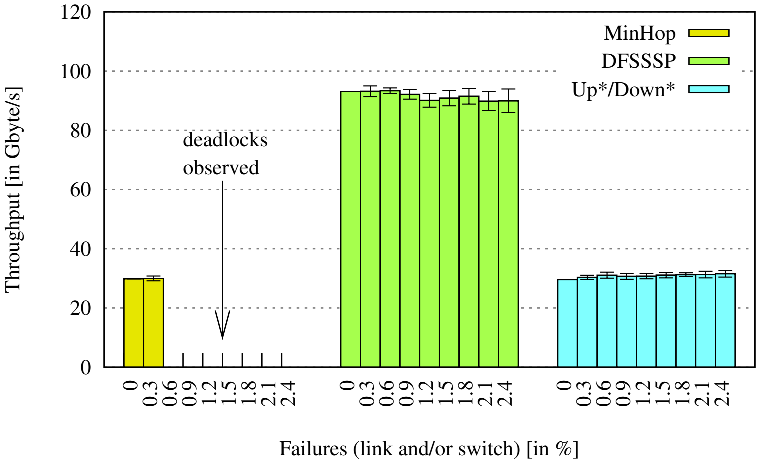
The conclusion derived from the results in Figure 4.8 and Table 4.6 is that a change from the default routing algorithm, which was MinHop, to the deadlock-free SSSP routing increased the throughput for exchange patterns by a factor of three. Furthermore, the change from MinHop to DFSSSP routing also removed the possibility of deadlocks in the InfiniBand fabric, which the simulations have experimentally verified for MinHop, see missing bars in the figure. Remarkable is the fact that a fail-in-place approach for the Deimos HPC system would have had almost no effect on the achievable performance of the communication layer, as shown by the almost unchanged throughput despite injected failure. Hence, maintenance costs for Deimos' network could have been saved completely, except for repairing node-to-



(a) Link failures only (0.2% annual failure rate)



(b) Switch failures only (1.5% annual failure rate)



(c) Switch and link failures (1 : 2 ratio)

Figure 4.8: Histograms for exchange patterns for different failure types using MinHop, DFSSSP and Up*/Down* routing on Deimos (LASH excluded because throughput is only 9% of DFSSSP's); Shown: mean value and 95% confidence interval for ten runs per failure percentage (seeds $\in \{1, \ldots, 10\}$); Deadlocks observed using MinHop routing for at least one out of ten seeds; Discussion in Section 4.4.5

Table 4.6: Intercept, slope, and R^2 for TSUBAME2 and Deimos shown in Figure 4.7c and Figure 4.8c (default routing: *italic*; best fail-in-place routing: **bold**)

HPC system	Routing	Intercept [in Gbyte/s]	Slope	R^2
TSUBAME2	**DFSSSP**	1,393.40	-1.33	0.62
	Fat-Tree	1,187.19	-1.48	0.66
	Up/Down**	717.76	-0.08	0.01
	LASH	22.92	-0.01	0.10
Deimos	*MinHop*	29.94	-	-
	DFSSSP	93.40	-0.15	0.09
	Up*/Down*	30.10	0.06	0.11
	LASH	8.37	0.00	0.04

switch failures or critical failures which would bisect the topology.

Overall, Section 4.4 has shown that HPC networks, and not only hard drives, can be operated in fail-in-place mode to reduce maintenance costs and to utilize the remaining resources. Even so the resulting irregular topologies pose a challenge to the used routing algorithms, a low failure rate of the network components supports a fail-in-place network design strategy, which bypasses non-critical failures during the supercomputer's lifetime.

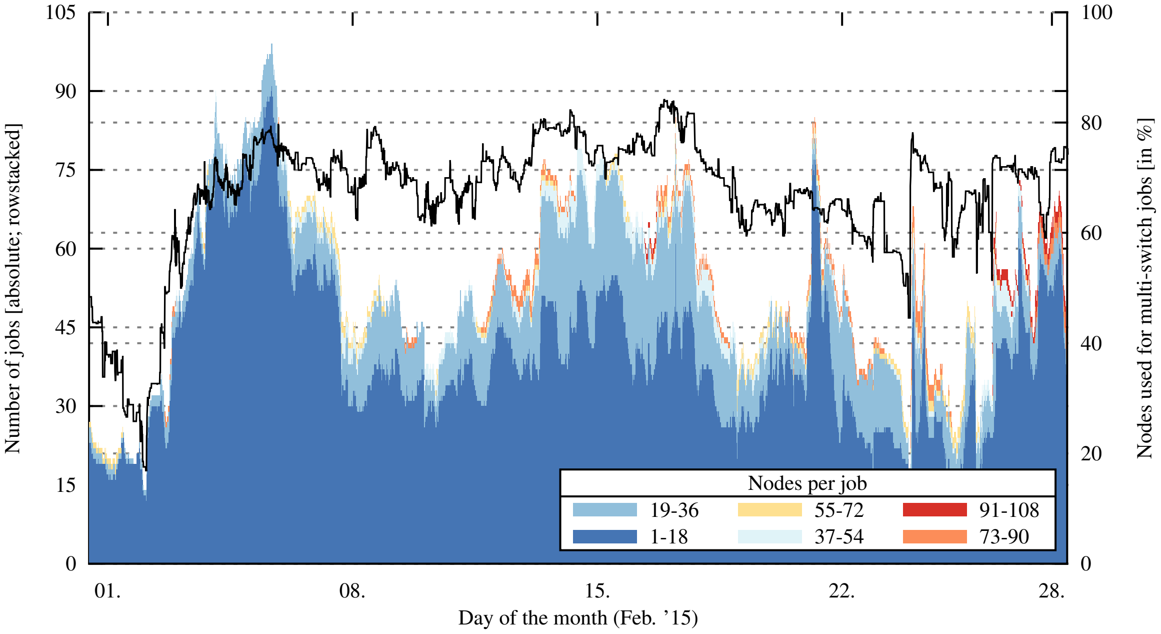
5 Utilization Improvement through SAR

Idealized conditions, e.g., regular topology without faulty network components, single application running on the whole system, or no system noise which is altering the message injection pattern, etc., usually do not apply to production supercomputers. This chapter is analyzing the disadvantages of application-agnostic static routing policies in Section 5.1. Based on this analysis, a novel scheduling-aware routing approach (SAR) is presented in Section 5.2 which aims at optimizing the utilization of the interconnection network as well as the communication performance of applications. Section 5.3 shows the practicality and effectiveness of this scheduling-aware routing with simulations of real-world workloads and through the usage on a production supercomputer.

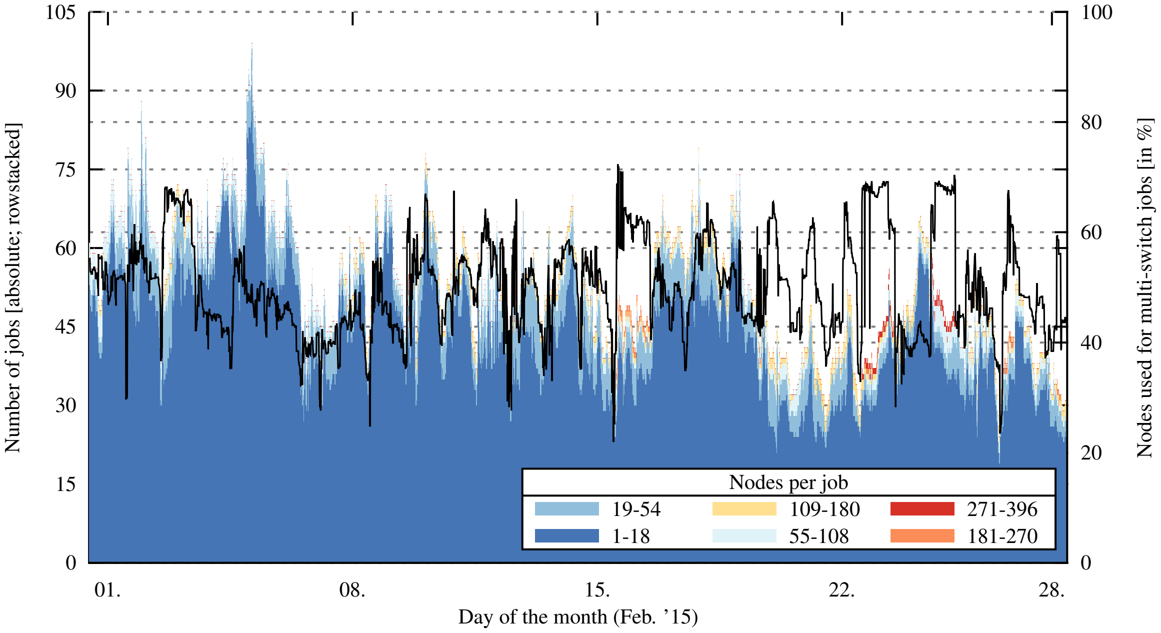
5.1 Conceptual Design of SAR for shared HPC Environments

Many advanced approaches for application-specific [Pal+06; Kin+09] or topology-specific [Sub+12; Pri+13b; Jok+15] routing and mapping assume idealized conditions such as a regular topology without faulty components, isolated bulk-synchronous applications communicating in synchronized phases, and the absence of system noise. Unfortunately, these assumptions are rarely true in practice (Chapter 4).

An analysis of the job mix on two production supercomputers emphasizes the complexity of real-world installations. Figure 5.1 shows all batch jobs that are using at least two network switches (which are referred to as multi-switch jobs hereafter) on two InfiniBand-based petascale supercomputers introduced in Section 3.2.6, i.e., Taurus and TSUBAME2, in the time frame of one month. These plots emphasize the fact that multiple parallel applications are spread throughout the system at any time and are thus contending for bandwidth on the shared network resources. In fact more than 66% and 50% of the compute jobs on Taurus and TSUBAME2, respectively, are using multiple switches. Thus, the jobs potentially share the same network resources, meaning switches and inter-switch links, and therefore can influence each others communication performance. But at the same time, communications do generally not cross from one parallel applications to another, with the exception of management and I/O traffic targeted at special endpoints in the system. Furthermore, the fact that these systems are used by multiple users simultaneously, submitting a broad set of scientific applications, results inevitably in a fragmentation, where the application processes are scheduled on compute nodes distributed all over the whole system. However, the batch systems usually prioritize system utilization over job locality. Other researchers, e.g., Jokanovic et al. [Jok+15], identified similar system fragmentations on their supercomputers, when the system is running continuously over a longer time period.



(a) Taurus supercomputer (multi-island design with 2014 compute nodes connected by FDR/QDR InfiniBand)



(b) TSUBAME2 petascale HPC system (utilizing 1408 compute nodes connected by two full-bisection fat-tree QDR InfiniBand networks)

Figure 5.1: Batch job history (sampled every 10 min) of two petascale HPC systems for one month of operation; Only multi-switch jobs shown; Total system utilization (inclusive single-node and single-switch jobs) is usually higher

When computing oblivious routes for an interconnection network, as all Algorithms of Section 3.3 do, the algorithm tries to balance the number of routes across all links so that each physical network cable has a similar number of routes crossing it. This number is called the edge forwarding index (EFI), see Definition 3.12, a theoretical upper bound for the worst-case congestion of a set of routes [HMS89]. The following experiment illustrates this concept: Route a two-level fat-tree network with an optimal oblivious strategy, i.e., with ideally balanced EFI, and schedule three different scientific applications (or batch jobs) to the system. The allocation of application processes to nodes logically partitions the network into three pieces which are not crossed by communications. Subsequently, the effective EFI for each link is computed by only counting routes connecting endpoints that belong to the same job. This metric extends the established edge forwarding index to differentiate between "useful" intra-job paths and paths connecting compute nodes which do not work on the same scientific problem, see Definition 5.1. In this regard, a batch job, or short job $j \in J$, is a (parallel) application scheduled onto a subset of compute nodes, $N_j \subseteq N$, which collectively work on a given task during a finite time frame.

Definition 5.1 (Effective Edge Forwarding Index)**.** *The **effective edge forwarding index** $\gamma^{e}_{R,J}$ of a switch port or outgoing link $c_q \in C^*$ is the sum of intra-job routes being forwarded via this port as induced by a given routing function R, i.e.,*

$$\gamma^{e}_{R,J}(c_q) := \sum_{j \in J} \left| \{ P_{n_x,n_y} \mid n_x, n_y \in N_j \wedge c_q \in P_{n_x,n_y} \} \right|$$

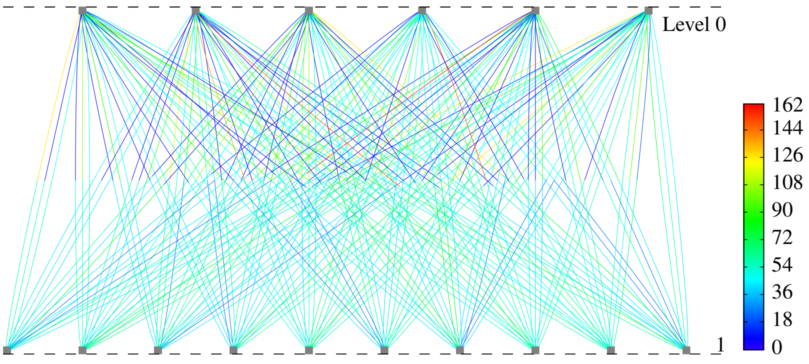
for all jobs $j \in J$ running simultaneously on the system.

Figure 5.2a shows a heat map of the effective EFIs for each link in the network. Each full-duplex link has two colors assigned, one color for the number of "useful" path forwarded via the respective egress port on the switch. Similarly, Figure 5.2c shows the corresponding histogram of the edge forwarding indices. Two conclusions can be directly derived from the figure:

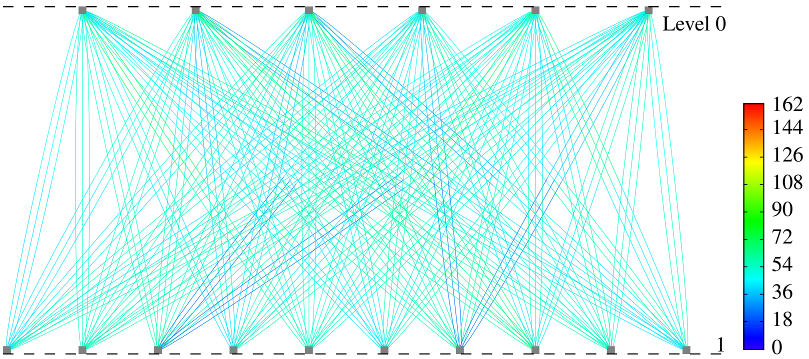
- There are hotspots of high route counts per link which can decrease communication performance of one or more jobs, and
- Some links are underutilized or not used at all which reduces the system's efficiency.

A less obvious observation is that the total number of switches and links available for intra-job communication is suboptimal.

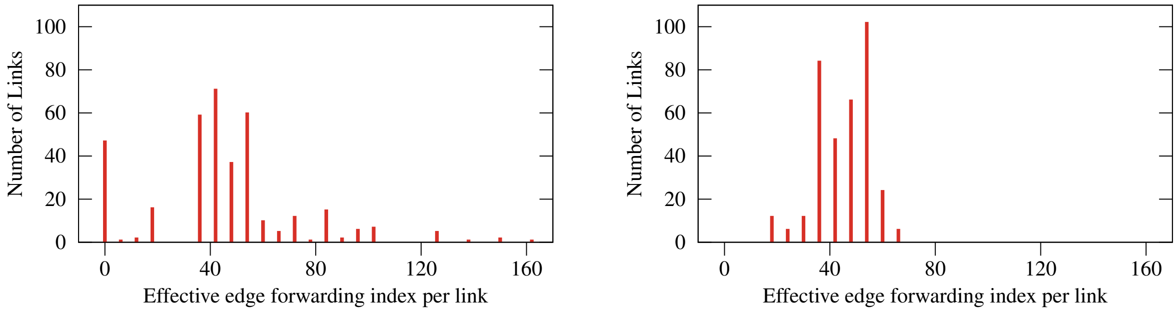
The same example can be used to illustrate the contributed (and subsequently introduced) novel approach of scheduling-aware routing (SAR) which aims at maximizing the allocation of network resources per batch job. The underlying assumption is that the allocated, and potentially shared, network resources per individual batch job should be maximized on a production system, which is contrary to other researcher's premise to isolate jobs to achieve performance predictability [Jok+15]. Hence, using SAR, intra-job communication throughput increases due to the increased available network hardware and this directly increases the utilization of the overall supercomputer. In other words, SAR exploits the knowledge of



(a) Oblivious routing



(b) Scheduling-aware routing



(c) Histogram for Figure 5.2a

(d) Histogram for Figure 5.2b

Figure 5.2: Comparison of effective EFI for inter-switch links (heat map in 5.2a and 5.2b); Oblivious routing vs. scheduling-aware routing for three equal-sized batch jobs on a 2-level fat-tree with 18 nodes attached to each of the ten switches of level 1; Jobs allocated using interleaving mapping (i.e., j_1 uses nodes 1 – 30 and 91 – 120, j_2 uses 31 – 60 and 151 – 180, and j_3 uses 61 – 90 and 121 – 150); Two colors of individual full-duplex links in 5.2a and 5.2b indicate different EFIs for the two opposite directions; Topology is equivalent to the full-bisection 180-node QDR island of Taurus (see Figure 3.6)

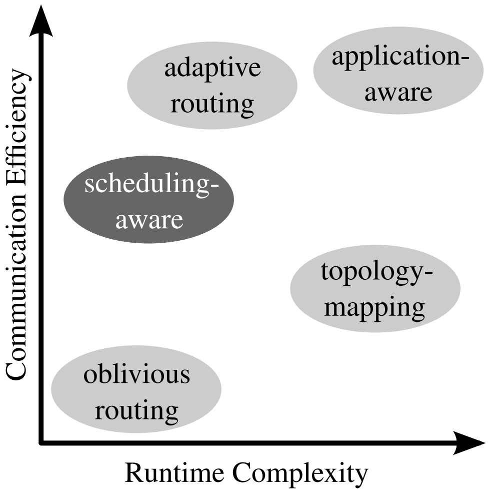


Figure 5.3: Qualitative comparison (with respect to runtime complexity of the approach and resulting communication efficiency) of different routing and network optimizations for supercomputers

the batch system, i.e., the current job mix, to determine the application locality and guides the fabric manager in assigning routing paths in the network. Furthermore, SAR should have a similarly low runtime complexity as other oblivious routing algorithms, and broad applicability to current and future interconnection technologies and supercomputers. Figures 5.2b and 5.2d show the same three-job example as before but using the scheduling-aware routing approach that this chapter introduces. The maximum EFI, i.e., upper bound for congestion, reduced from more than 160 to a mere 60 and the overall path balance is greatly improved.

A second metric, derived from the previously defined effective edge forwarding index, describes the amount of superfluous links in the network, metaphorically speaking: the proportion of "dark fiber" to links usable for intra-job communication, see Definition 5.2. Please note, that this latter metric does not differentiate between copper or fiber cables, and fiber is used as a synonym for link. Furthermore, this metric is only based on intra-job paths, and ignores any other traffic related to supplementary nodes, such as I/O traffic to and from a remote filesystem or administrative data paths.

Definition 5.2 (Dark Fiber Percentage)**.** *The* ***dark fiber percentage*** *is the percentage of links in the system, which are not used for intra-job routes, and can therefore be derived from* $\gamma^{e}_{R,J}$ *in the following way:*

$$\theta_{R,J} := \frac{\left|\{c_q \in C^* \mid \gamma^{e}_{R,J}(c_q) = 0\}\right|}{|C^*|}$$

The analyses in Section 5.3 utilize the dark fiber percentage, among with the effective edge forwarding index, to evaluate the effectiveness of the SAR approach compared to other state-of-the-art oblivious routing strategies. A general overview and qualitative comparison of the different network/communication optimization strategies is shown in Figure 5.3.

5.2 Example Implementation of the Scheduling-Aware Routing

The previous section sketched the idea and potential advantages of a scheduling-aware routing for supercomputers which is simultaneously used by multiple users. The realization of this approach will be demonstrated in the following sections. Section 5.2.1 outlines the basic components, including relevant features and limitations, available on the petascale Taurus HPC system, which was available as a testbed for SAR. Section 5.2.2 provides details on how these building blocks are extended and combined to implement the scheduling-aware routing (SAR). Finally, Section 5.2.3 introduces a novel protocol to prevent out-of-order packet delivery, potentially caused by SAR's frequent network reconfigurations.

5.2.1 Hardware and Software Building Blocks

The main building blocks, which are needed to implement the SAR approach, are subject of this section. Firstly, the essential details of the inner workings of InfiniBand and its subnet manager at presented in Section 5.2.1.1. Furthermore, the Sections 5.2.1.2 and 5.2.1.3 provide a brief introduction into the SLURM batch system and Open MPI communication library, respectively.

5.2.1.1 InfiniBand and OpenSM

The InfiniBand technology [Pfi02], as defined by the architecture specification (IBTAspec) [Inf15], is widely used in current HPC systems to network compute and storage nodes. Each network device in the fabric is identified by one or more local identifiers (LIDs), configured via a lid mask control (LMC) parameter, such that the total number of LIDs per device is 2^{LMC}, with $0 \leq \text{LMC} \leq 7$.

The IBTAspec defines a layer, called verbs, between the software/operating system and the InfiniBand hardware, see Figure 29 of [Inf15, p. 130]. Communication between two IB devices is implemented via queue pairs (QPs) that establish a send and receive channel between one source LID and one destination LID for unicast traffic. Multicast traffic is beyond the scope of this thesis, because it is performed unreliably in InfiniBand [Inf15, 7.10.2], i.e., by using the unacknowledged Unreliable Datagram transport service, and therefore generally not used by parallel applications of the HPC community. However, the verbs layer also allows modifications of QPs, e.g., performing path migrations (APM) or draining all outstanding work requests (SQD). These two features will be used in Section 5.2.3 to enforce property preserving network updates while changing the routing configuration.

Applications or communication libraries, such as Open MPI, post work requests (WRs) to a queue pair to initiate data transport to a remote node. InfiniBand's hardware offloading and kernel-bypass characteristics allow the processing of these WRs asynchronously to the application. The channel adapter, i.e., network interface card of InfiniBand, splits a scheduled message into packets, configures packet headers, and dispatches them. Achieving per-flow consistency is made more complicated because of this asynchronicity. Section 5.2.3 will elucidate this issue further. InfiniBand ensures lossless processing of WRs for reliable channels through credit-based flow control, both on a per-link basis and end-to-end

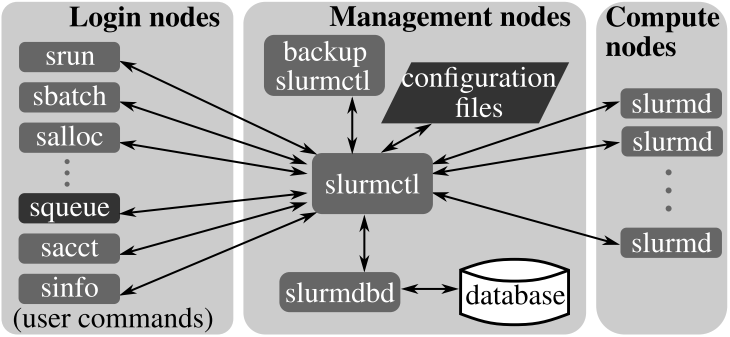


Figure 5.4: SLURM architecture; Highlighted: squeue and configuration files which will be used by the filtering tool in Section 5.2.2.1

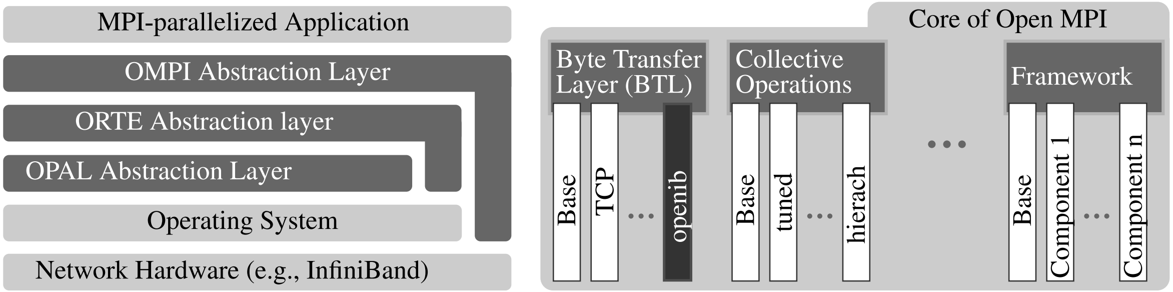
between send/receive queues. However, reliable channels require in-order delivery of all packets of one WR, otherwise the receiving IB port drops messages and requests retransmissions.

Moreover, IB supports virtual lanes (VLs), which are basically different sets of buffers within one port. Hence, InfiniBand's VLs are similar virtual channels, and therefore can be used to construct virtual layers, see Definition 3.8. InfiniBand network ports must support up to 15 VLs for data traffic, indexed starting from 0, and one virtual lane for management traffic, i.e., VL15. However, current hardware usually supports only four or at most eight data VLs. Data virtual lanes can be used for quality of service or to avoid potential deadlock configurations in the forwarding tables of switches, as described in Section 3.1.2. Various routing approaches use this feature to combine the i^{th} VL of each port into virtual layers and assign paths to different layers to create a deadlock-free routing configuration [SLT02; Shi+09; DHN11].

The InfiniBand fabric is controlled by a subnet manager, which is responsible for assigning LIDs and LMCs to nodes, reacts to topological changes and failures in the fabric, and calculates the routing configuration for the fabric, among other things. An open-source version of this subnet manager is called OpenSM [Mel13], which currently implements nine flow-oblivious routing algorithms, see Sections 3.3.1 to 3.3.7, for a variety of supported network topologies. Chapter 4 unveils feasible combinations of these routing algorithms and (faulty) topologies. Once a new routing configuration is calculated, OpenSM updates the linear forwarding tables of all IB switches, and packets are forwarded according to these new switching rules. An LFT update usually only happens in response to a topological change in the fabric, e.g., when network components fail or new nodes are being added. Definition 3.5 details forwarding tables formally.

5.2.1.2 Simple Linux Utility for Resource Management

Many supercomputers utilize SLURM [YJG03] as a cluster resource manager and batch system. SLURM is open-source and offers a plugin interface, which makes it a preferred solution for many HPC sites, since it offers the extensibility for individual needs and it offers the option to test research ideas. The SLURM architecture, see Figure 5.4, primarily consists of three parts: (1) user commands, (2) controller daemon, and (3) compute node daemons. The controller daemon manages most of the system's compute resources,



(**a**) Abstraction layer (**b**) Frameworks and components

Figure 5.5: Open MPI's modular component architecture establishes a message passing interface between application and hardware to enable inter-process communication; Highlighted: openib BTL for InfiniBand

and schedules batch jobs onto free compute nodes. However, the controller is partially or fully unaware of other resources, such as remote storage or interconnection network. For example, while SLURM is aware of the physical topology of the supercomputer, i.e., node-to-switch mapping and inter-switch connectivity, it is unaware of the deployed routing configuration by the subnet manager. The SLURM controller can be attached to a database, see Figure 5.4, which stores the historical and current state of job allocations for a system. In Section 5.2.2, this information is used to obtain a snapshot of simultaneously running jobs and their locality.

5.2.1.3 Open MPI

The message passing interface (MPI) specification [Mes15] standardizes an API for parallel applications to perform inter-process communication, either locally on a compute node or via the interconnection network. One widely used open-source implementation of the MPI standard is Open MPI [Gab+04]. The Open MPI library assigns one rank, with rank $\in \{0,\ldots,n-1\}$, to each processes within a single MPI-parallelized application, and therefore provides an abstract interface to the application to perform point-to-point or collective operations between ranks. Open MPI uses a modular component architecture (MCA) to support a variety of interconnection technologies and transport services, see Figure 5.5. The OMPI layer provides the MPI API to the applications and implements communication algorithms for point-to-point and collective operations, monitoring and profiling interfaces, etc. One framework within the MCA is the byte transfer layer (BTL), responsible for point-to-point communication between MPI processes. The BTL for IB networks is called "openib", see Figure 5.5b. This openib component interfaces with the verbs layer, see Section 5.2.1.1, to set up QPs, schedule send or receive WRs, and handle events raised by the InfiniBand hardware, among other things. When Open MPI is configured with threading support, i.e., for POSIX threads, then the openib component spawns one asynchronous event thread per MPI process. In Section 5.2.3, this asynchronous event thread is used to establish communication channels between OpenSM and MPI programs to achieve per-flow consistent network updates.

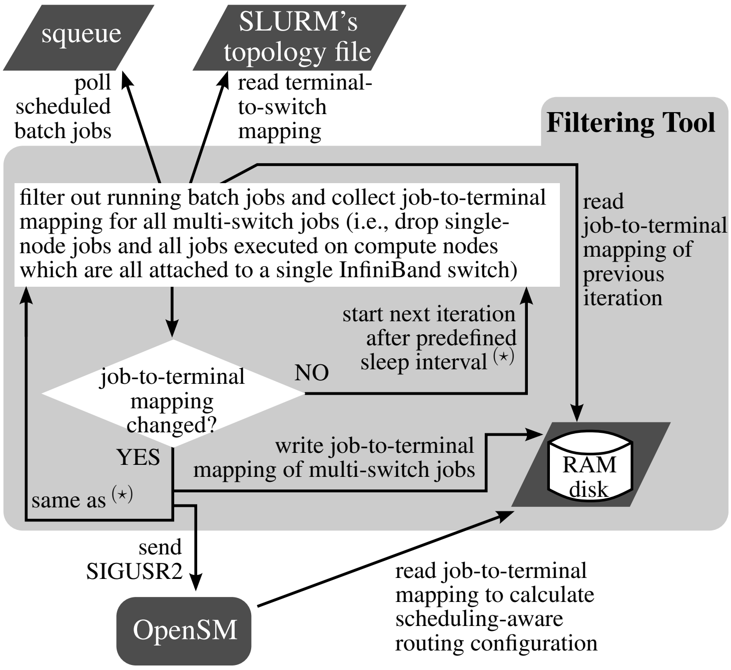

squeue
SLURM's topology file
poll scheduled batch jobs
read terminal-to-switch mapping
Filtering Tool
filter out running batch jobs and collect job-to-terminal mapping for all multi-switch jobs (i.e., drop single-node jobs and all jobs executed on compute nodes which are all attached to a single InfiniBand switch)
read job-to-terminal mapping of previous iteration
start next iteration after predefined sleep interval (⋆)
NO
job-to-terminal mapping changed?
YES
write job-to-terminal mapping of multi-switch jobs
RAM disk
same as (⋆)
send SIGUSR2
OpenSM
read job-to-terminal mapping to calculate scheduling-aware routing configuration

Figure 5.6: Flowchart of a filtering tool; Collect information about currently running multi-switch batch jobs and initiate recalculation of LFT via SAR

5.2.2 Scheduling-Aware Routing Optimization

This section introduces the novel scheduling-aware routing that this thesis contributes. SAR optimizes the forwarding tables for production supercomputers by filtering relevant information from the batch system and passing this information to a modified version of the DFSSSP routing engine.

5.2.2.1 Filter between Batch System and Subnet Manager

Performing a scheduling-aware routing optimization for a supercomputer is only beneficial for parallel applications when their compute nodes are attached to multiple switches. Therefore, a filtering tool optimizes the workflow, see Figure 5.6, whereby it periodically polls SLURM for the current state of the system. To be exact, information about batch jobs, including job state (pending, running, etc.), job identifier, and allocated compute nodes, are collected. This information is obtainable through SLURM's squeue command. Furthermore, the topology information is required, i.e., terminal-to-switch mapping, which is stored among SLURM's configuration files, or alternatively can be obtained by ibnetdiscover, an IB tool [Mel13]. The filtering tool uses this topology information to assemble a list of multi-switch jobs containing all currently running applications which have their compute nodes attached to at least two switches. The list of multi-switch jobs is compared to the list of the previous iteration while ignoring the actual job identifiers. Hence, the filtering tool can avoid unnecessary calculations of LFTs, since replacing one job with another job on the same compute nodes will not result in a different routing configuration while using SAR, see subsequent Algorithm 5.1. If the two lists of multi-switch jobs differ, then the filtering tool writes a job-to-terminal mapping file for OpenSM, informs OpenSM about the configuration change, and replaces the previous list with the new list for the next iteration. Additionally, an extension to OpenSM's signal handler is a requisite to process the SIGUSR2 signal to inform the OpenSM about an updated job-to-terminal mapping.

A direct interface between SLURM and OpenSM would have been possible to, but is less desired since: First off, portability, to easily support other batch systems or subnet managers, such as the Portable Batch System (PBS) [Hen95] and derivatives, or the fabric manager for Intel Omni-Path [Bir+15]. Furthermore, the SLURM controller and OpenSM can run on different management nodes of an HPC system. Lastly, integrating the filtering functionality into the SLURM controller potentially increases the latency experienced by users when submitting many jobs due to an additional computational overhead.

5.2.2.2 Routing Optimization with DFSSSP

The basis of SAR is the deadlock-free single-source shortest-path routing (DFSSSP) for three reasons: (1) DFSSSP is deadlock-free and topology-agnostic, and therefore supports a wide variety of network topologies [DHN11], (2) DFSSSP offers high global throughput for the complete HPC system and offers a well-balanced EFI across the system, see Chapter 4, and (3) DFSSSP already distinguishes three terminal types, i.e., compute, storage, and others nodes, and optimizes their routing separately. Hence, SAR inherits the characteristics listed in (1) and (3) and improves upon (2).

DFSSSP routing computes the destination-based paths for the nodes of the interconnection network by applying a modified version of Dijkstra's shortest-path algorithm, optimized for multigraphs, see Sections 3.3.2 and 3.3.8. The routes from all nodes to one destination in the network are derived by reversing the paths calculated from Dijkstra's source to all vertices in the graph. DFSSSP iterates over all nodes of the network, and firstly applies Dijkstra's algorithm, followed by an update of the edge weights in the graph. This edge weight update, increasing the weight by $+1$ for every new path on a link, ensures the well-balanced EFI, which is mentioned above in reason (2). After calculating a path for every node-to-node combination, DFSSSP assigns these paths into virtual layers to achieve a deadlock-free routing configuration, similar to other routings, such as LASH [SLT02].

The extensions for DFSSSP routing to include knowledge about batch job locations, and enable the algorithm to optimize the path calculation for intra-job paths, is shown as pseudo code in Algorithm 5.1. The main modifications are outlined in lines $1-10$, while subsequent lines are shown for sake of completeness and sketch the remainder of the already existing DFSSSP implementation in OpenSM. In the first part of Algorithm 5.1, the job-to-terminal mapping given by the filtering tool is analyzed, see Section 5.2.2.1. Each node in the multigraph, representing the interconnection network, gets extended by a job array which stores all jobIDs currently running on this node. These nodes are sorted descending by the size of largest job (in terms of node count) executed on the node, i.e., compute nodes belonging to the largest job running on the HPC system will be processed first in the following loop, see line 6. Hence, this sorting ensures that intra-job path balancing is increased and the number of overlapping

Algorithm 5.1: Scheduling-aware DFSSSP routing

```
   Input: Network $I = G(N, C)$
          Job-to-terminal mapping $B := [(\text{nodeName}, \text{jobID}), \ldots]$
   Result: Scheduling-aware and deadlock-free routing configuration
           ($P_{n_x,n_y}$ for all $n_x, n_y \in N$)
   /* Process job-to-terminal mapping                                              */
1  foreach node $n \in N$ do
2      $n$.jobList ← empty list []
3      foreach pair (nodeName,jobID) $\in B$ do
4          if $n.nodeName = nodeName$ then n.jobList.append(jobID)
   /* Compute optimized routing for compute nodes                                  */
5  $N_{\text{sorted}}$ ← Sort $N$ descending by the job size executed on $n \in N$
6  foreach node $n_d \in N_{sorted}$ do
7      Calculate one path $P_{n_x,n_d}$ for every pair $(n_x, n_d)$, with $n_x \in N$, with the modified Dijkstra algorithm (details in [DHN11])
8      foreach node $n_x \in N$ do
9          if $n_x.jobList \bigcap n_d.jobList \neq \emptyset$ then
10             Increase edge weight by $+1$ for each link in path $P_{n_x,n_d}$
   /* Reset edge weights and compute optimized routing for storage nodes           */
11 foreach storage node $n_d \in N$ do
12     Calculate one path $P_{n_x,n_d}$ for every pair $(n_x, n_d)$, with $n_x \in N$, with the modified Dijkstra algorithm
13     Update edge weights for all links used by $P(\cdot, n_d)$
   /* Reset edge weights and compute optimized routing for all other nodes         */
14 foreach node $n_d \in N \wedge n_d$ not processed before do
15     Calculate one path $P_{n_x,n_d}$ for every pair $(n_x, n_d)$, with $n_x \in N$, with the modified Dijkstra algorithm
16     Update edge weights for all links used by $P(\cdot, n_d)$
   /* Create deadlock-free routing configuration                                   */
17 foreach route $P_{n_x,n_y}$ calculated above do
18     Assign $P_{n_x,n_y}$ to one virtual layer without creating a cycle in the corresponding channel dependency graph (see [DHN11] for details)
```

paths is minimized. Since edge weights are only updated when actually used by an intra-job path, see lines 9 and 10 of Algorithm 5.1, the resulting paths should improve the network metrics compared to other state-of-the-art oblivious routings, which will be demonstrated in Section 5.3. The rest of the base DFSSSP algorithm is kept in place to ensure deadlock-freedom and a well-balanced routing configuration towards the remote filesystem.

5.2.3 Property Preserving Network Update for InfiniBand

One problem while performing network updates, i.e., the transition between two network states, is to preserve per-packet or per-flow consistency [Rei+12; McC+15], and therefore preserve certain properties satisfied by each network state. Usually, “atomic” state transitions for the whole network are impossible, hence without consistency, these network updates can lead to security vulnerability, loss of packets, etc. A summary of the comprehensive theoretical groundwork given by Reitblatt et al. [Rei+12] is outlined in the following two Definitions 5.3 and 5.4.

Definition 5.3 (Per-packet and Per-flow Consistency)**.** ***Per-packet consistency*** *of the network operation is achieved if and only if each packet injected into the network is processed by exactly one network state, see Definition 3.6. If all packets of each injected message or flow are processed by the same network state, then this is referred to as* ***per-flow consistency****.*

Definition 5.4 (Per-packet Consistent Network Update)**.** *Let u_s be a sequence of atomic updates u applied to a set of network switches. Then, for two (distinguishable) network states χ and χ', the transition $\chi \rightarrow_{u_s} \chi'$ is called a* ***per-packet consistent update*** *if and only if each packet is processed by either χ or χ', but not any inconsistent state in-between.* ***Per-flow consistent updates*** *are defined analogously.*

The property preservation of network updates, as stated by Theorem 1 of Reitblatt et al. [Rei+12], is a direct result when per-packet consistent updates are performed. However, this only holds for properties related to single packets, e.g., connectivity or loop-freedom. Other network properties, such as in-order delivery or deadlock-freedom, which are based on the correlation of multiple packets are harder to achieve. Hence, for the purpose of scheduling-aware rerouting of HPC systems, Reitblatt's property preservation needs to be extended to be applicable to lossless and deadlock-free interconnects, see Definition 5.5.

Definition 5.5 (Property Preserving Network Update)**.** *The transition $\chi \rightarrow_{u_s} \chi'$ between two network states χ and χ' is called a* ***property preserving network update*** *for a lossless interconnection network if the following conditions hold:*

1. *Both network states, χ and χ', are based on deadlock-free routing configurations,*
2. *$\chi \rightarrow_{u_s} \chi'$ is a per-flow consistent update for reliable channels of communication, and per-packet consistent update for unreliable connections, and*
3. *A temporally simultaneous processing of packets, either by χ or χ', during a transition cannot cause a deadlock.*

Attribute 2 of Definition 5.5 can be weakened to per-packet consistent updates when in-order delivery is not required for reliable connections. Assuming the scheduling-aware (re-)routing is a property preserving network update, then a fault-free execution of parallel applications, with respect to network-related problems, is guaranteed, and therefore an uninterruptible operation of the HPC system. Further network properties based on multiple packets, such as congestion control or quality of service, are beyond the scope of this thesis.

Changing the routing configuration of an InfiniBand network cannot be done atomically. OpenSM, the subnet manager of InfiniBand, splits the LFT of each switch into multiple management datagram (MAD) packets, each carrying a payload of 64 bytes, to distribute new LFTs to all switches. Hence, at most 64 forwarding rules per switch can be changed at once, because 8 bit are used to encode an egress port. This non-atomicity can cause out-of-order packet delivery for a flow, and therefore preventing per-flow consistency and disrupting InfiniBand's reliable connection service, see Section 5.2.1.1, resulting in application crashes in the worst case.

Multiple network update protocols for software defined networks (SDN) have been proposed [Rei+12; McC+15]. For example, the proposed two-phase updated installs passive routing configurations which are activated when the switch identifies a certain packet, so that subsequent packets follow the same path. Unfortunately, none of these protocols are applicable to InfiniBand due to missing features, i.e., the ability to deploy two routing configurations simultaneously, to distinguish between flows, or to tag packets. Hence, updating the forwarding rules of a path in InfiniBand is only safe when the used method can guarantee that no packet or flow uses the entire path during the update process.

A feasible and novel solution to the above outlined problem is the five-phase update protocol, introduced hereafter, which is able to guarantee property preserving network updates for reliable connections, see Definition 5.5, within the limitations of the InfiniBand architecture specifications (IBTAspec) [Inf15]. Open MPI in version 1.8.4 and OpenSM in version 3.3.18 are used for the prototype implementation of this update protocol. However, this protocol can be applied to other services in a supercomputer, e.g., for reliable communication with a remote filesystem. The sequence diagram in Figure 5.7 exemplifies the interaction between an MPI-parallel application and OpenSM, which performs the routing reconfiguration.

The initial assumptions for this update protocol to be applicable are the following, which will ensure deadlock-freedom prior to the network update:

- OpenSM assigns two LIDs, while distinguishing between baseLID and highLID (the latter equals to baseLID $+ 1$), for each node in the network via LMC $= 1$,
- The routing configuration for baseLIDs is calculated by Algorithm 5.1 (using VLs $0, \ldots, n-2$),
- The routing configuration for highLIDs is calculated by Up*/Down* and uses VL $= n-1$, and
- Packets are only sent from baseLIDs to baseLIDs or among highLIDs, but not a mixture of both.

During phase 1 of the five-phase update protocol shown in Figure 5.7 an MPI-parallelized application, or the asynchronous event thread (AsyncThread; see Section 5.2.1.3) of rank 0 to be precise, uses InformInfo MAD packets to subscribe for event forwarding, as outlined in IBTAspec [Inf15, 13.4.8.3, 13.4.11]. All packets for inter-process communication within the MPI application are using baseLIDs for destination addressing. The MPI library uses a library of the IB software stack to send userspace management datagram (uMAD) packets to communicate with the OpenSM. The uMAD library also allows the AsyncThread of rank 0 to periodically poll for MAD packets send by the OpenSM. The AsyncThread subscribes only for unpath and repath traps, see IBTAspec Section 14.4.12, which are part of the general concept of asynchronous notifications or alerts between IB components. Theoretically, subscriptions and forwarded traps contain individual paths records, however the subscriptions used here apply for the whole subnet's LID range, and traps contain empty path records. This increases scalability by reducing the number of MADs handled by both sides, i.e., only one MAD, with either unpath or repath trap, has to be sent out to each subscriber. These traps are intended to inform terminals about usability changes of certain routes. Once the OpenSM calculates a new routing configuration for the baseLIDs, in consequence of the approach outlined in Section 5.2.2, it raises an unpath trap and forwards the trap to all subscribers, but it withholds to reconfigure any LFT. The AsyncThread of rank 0 forwards the unpath trap to all other ranks of the same application by the use of MPI calls, i.e., primarily MPI_Isend, MPI_Irecv, and MPI_Test. This tree-like approach, where the OpenSM informs each rank 0 of all running MPI applications which

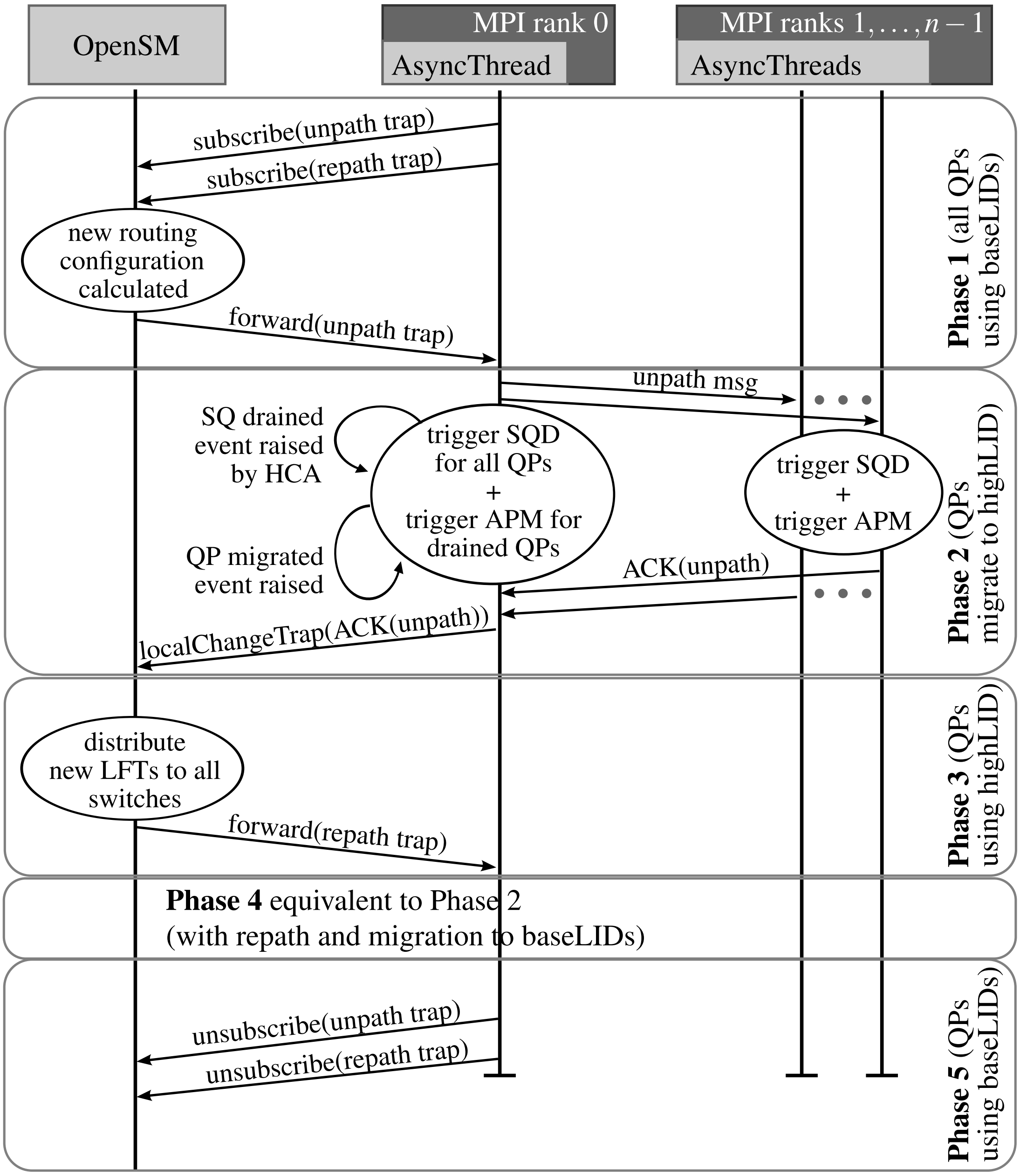

OpenSM
MPI rank 0
AsyncThread
MPI ranks 1, ..., $n-1$
AsyncThreads
subscribe(unpath trap)
subscribe(repath trap)
new routing configuration calculated
forward(unpath trap)
Phase 1 (all QPs using baseLIDs)
unpath msg
SQ drained event raised by HCA
trigger SQD for all QPs + trigger APM for drained QPs
QP migrated event raised
trigger SQD + trigger APM
ACK(unpath)
localChangeTrap(ACK(unpath))
Phase 2 (QPs migrate to highLID)
distribute new LFTs to all switches
forward(repath trap)
Phase 3 (QPs using highLID)
Phase 4 equivalent to Phase 2
(with repath and migration to baseLIDs)
unsubscribe(unpath trap)
unsubscribe(repath trap)
Phase 5 (QPs using baseLIDs)

Figure 5.7: Sequence diagram of a five-phase update protocol to achieve property preserving network updates of InfiniBand networks

then forward the traps internally, is required, because the uMAD library cannot be used simultaneously by multiple asynchronous event threads on one compute node.

In phase 2, all processes of all subscribed parallel applications modify their queue pairs, which belong to reliable connections, to enter the draining state by triggering the SQD event, see IBTAspec Section 10.3.1.5. All send work request posted prior to the draining, and all receive work requests are processed normally. While in draining state, MPI processes can still post new WRs to the verbs layer, however these will not be send out before the QP reenters the “ready-to-send” state. Once the draining of an QP is complete, the IB HCA sends an event notification to the AsyncThread. After receiving this “QP is drained”-notification, the AsyncThread triggers a path migration (APM) to the highLIDs, see IBTAspec Section 10.4, changes the QP back into the “ready-to-send” state, and then loads a new alternate path. The new alternate path is always the opposite of the current path, i.e., if the current path is baseLID→baseLID, then the new alternate path will be highLID→highLID. When all local QPs have been migrated to the highLID, then each process informs rank 0, and acknowledges the successful migration. The AsyncThread of rank 0 uses a “local changes” trap, see IBTAspec Section 14.3.13, informing the OpenSM about the successful draining of baseLID-related traffic, after it received acknowledgments by all ranks.

When the OpenSM receives confirmation by all subscribers, indicating that no reliable connections are using baseLIDs in the entire network, the protocol switches to phase 3. OpenSM reconfigures all switches with new LFTs for the baseLIDs, which were calculated during phase 1. The routing configuration for the highLIDs is never changed, since it relies on the oblivious Up*/Down* routing. Theses routes for highLIDs are kept in place until a failure in the network requires fault-recovery and rerouting for baseLIDs and highLIDs. Property preserving network updates during fault-recovery in an InfiniBand network is beyond the scope of this thesis.

Phase 4 starts after all InfiniBand switches have been reconfigured, and it is essentially a copy of phase 2 with the exception that the OpenSM requests the subscribers to migrate all their reliable connections from the highLIDs back to the baseLIDs. During phase 5, the subnet manager listens for applications canceling their forwarding subscriptions and waits for new scheduling-aware rerouting requests. Upon completion of the MPI-based application the asynchronous event thread of rank 0 sends out unsubscribe InformInfo MADs to the OpenSM.

The whole draining and path migration process is transparent to the overlying application. At most, the application could experience a minor latency increase for a particular QP while it is draining, and experience reduced throughput while the highLIDs are used.

5.3 Evaluation

This evaluation is split into three separate parts. To begin with, Section 5.3.1 provides a list of shortcomings of the prototype implementation of the five-phase update protocol due to current circumstances. The subsequent Sections 5.3.2 and 5.3.3 assess the advantages of the scheduling-aware routing approach in comparison to other flow-oblivious routing approaches.

5.3.1 Current Limitations and Problems

When this research was conducted, minor problems have been experienced while implementing the five-phase network update protocol, shown in Figure 5.7. For the sake of completeness, the following list mentions these limitations without going into much detail about possible solutions.

First off, the communication between the asynchronous event thread of rank 0 and the OpenSM is potentially subject to packet loss. The reason are the (u)MAD packets to subscribe and forward traps which must use QP0 and QP1. These special QPs are configured for unreliable transport service, see IBTAspec Sections 3.9.4 and 3.9.5. However, both the uMAD library and OpenSM usually send MADs multiple times if they are not acknowledged in time. Whenever the OpenSM reaches the maximum number of retries without receiving an acknowledgment, it assumes an erroneous termination of application and unsubscribes it from the internally stored list of subscribers.

Second, Open MPI combined with the openib component does not support simultaneous calls to the MPI API from multiple threads of one process. Manually serializing MPI calls between the main application and the AsyncThread via a pthread mutex lock is the currently chosen workaround.

And lastly, any attempts to modify QPs into the draining state was prevented by the tested firmwares for the IB devices of the testbed. Depending on firmware version, either the verbs call was possible but had no effect, or the firmware rejected the verbs call.

Therefore, the following sections omit showing practical results of the property preserving network update protocol on the petascale Taurus HPC system. However, the entire rerouting and update protocol, as outlined in Sections 5.2.2 and 5.2.3, has been tested successfully (apart from the QP draining) on a smaller testbed, which will be demonstrated in Section 5.3.3.5.

5.3.2 Theoretical Evaluation of Network Metrics

The evaluation and comparison of the effective edge forwarding index and dark fiber percentage, see Definitions 5.1 and 5.2, is based on the same batch job history which is presented in Figure 5.1 for the two HPC systems, Taurus and TSUBAME2. Hence, the toolchain of Chapter 4 is used to “exactly replay” the same job composition of each system, sampled every 10 min, and investigate the effects of different routing algorithms. These algorithms are Up*/Down*, fat-tree, and DFSSSP, all implemented in OpenSM, which are compared against the SAR implementation. The contestants are chosen based on the results of Section 4.4.4. As mentioned in Section 3.2.6, the vendor of the Taurus HPC system originally suggested the usage of fat-tree routing, before SAR got deployed in the course of this research, see Section 5.3.3. Furthermore, fat-tree routing is the current default on the TSUBAME2 supercomputer.

5.3.2.1 Effective Edge Forwarding Index

This section analyzes two metrics with respect to the EFI. The first metric is $\gamma^e(I,R,J) := \max_{c_q \in C^*} \gamma^e_{R,J}(c_q)$, i.e., the maximum $\gamma^e_{R,J}$ of all links, which implies hotspots of links shared by multiple jobs. This metric is similar to the EFI of the interconnection network, see Definition 3.12, however constrained to "useful" routes for jobs. Equivalently, the second metric is the maximal edge forwarding index $\gamma^e(I,R,j)$ per job j, which should also be as low as possible, because it bounds the worst-case congestion within a job, i.e., the link with the most intra-job routes crossing it.

The results for $\gamma^e(I,R,J)$ are shown in the top plot of Figures 5.8a and 5.8b. As one can see, the scheduling-aware routing (blue line) outperforms DFSSSP for the Taurus system and outperforms the other three routing algorithms on the TSUBAME2 supercomputer. A summary of the results is also provided in Table 5.1 for Taurus and in Table 5.2 for TSUBAME2 to ease a quantitative comparisons. The results in the two tables show the improvement by SAR for each metric (given as maximum and as average across all sample points within the investigated month, as well as the improvement in percent). For example, the 279.0 and 57.3 for DFSSSP mean that the scheduling-aware routing reduced $\gamma^e(I,\text{SAR},J)$ by 279.0 (or 50.8%), at least once compared to DFSSSP routing, and reduced $\gamma^e(I,\text{SAR},J)$ by 57.3 (or 23.3%), on average during the full month.

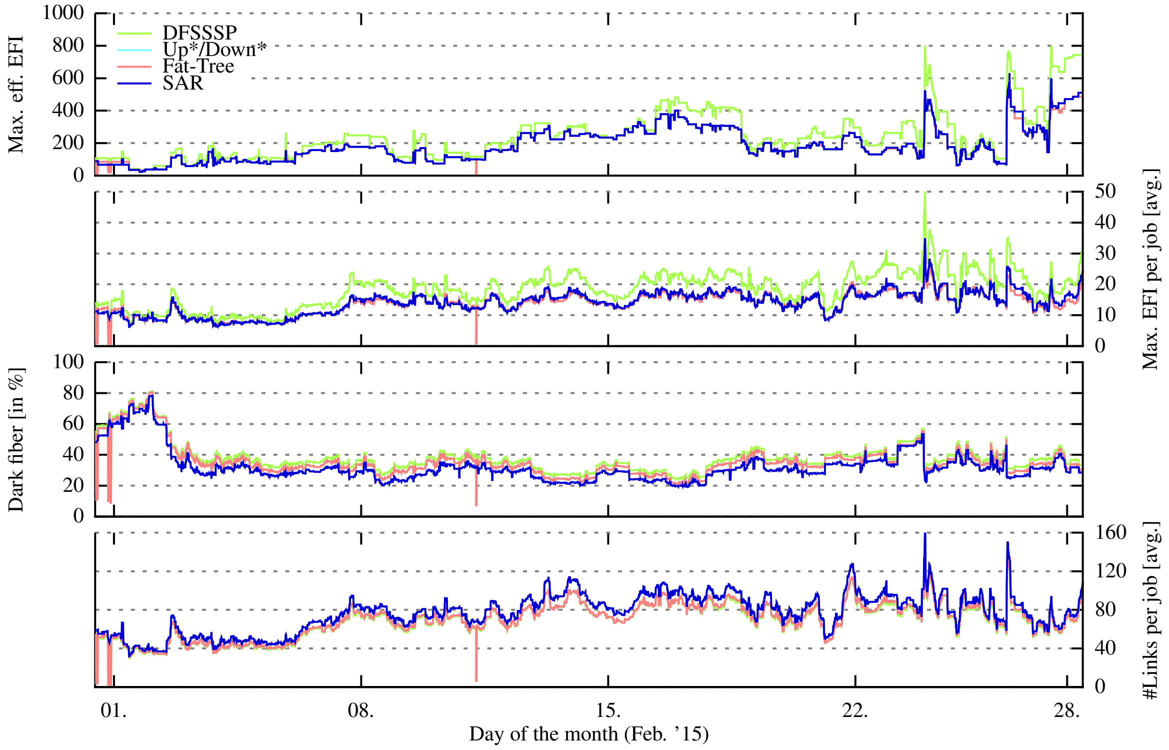
The figures show a reduction of $\gamma^e(I,R,J)$ when SAR is used for both HPC systems, especially for TSUBAME2. This indicates that the worst-case link congestion is lowered. The tremendous differences between fat-tree and the scheduling-aware routing on TSUBAME2, around day 16 and day 28, indicate very unfortunate job-to-node placements, which does not match the default fat-tree routing of TSUBAME2. The day 16 outlier will be analyzed in more detail in Section 5.3.2.3. SAR slightly increases $\gamma^e(I,R,J)$ on Taurus in comparison to fat-tree and Up*/Down* routing. However, on average SAR assigns only one more path onto the link with the highest load. This disadvantage in the worst-case bound of SAR is negligibly small, especially in the context of the other two metrics, link utilization and link availability. Please note that Taurus had a fault-free and regular topology during the investigated month and a valid assumption is that SAR will outperform fat-tree and Up*/Down* if the system's network topology becomes more irregular over time due to network faults, see Chapter 4.

For the second metric, the maximal edge forwarding index per job (2^{nd} plot of Figure 5.8) follows a similar behavior. The SAR approach lowers the maximum number of inter-job routes compared to the other routings, which should accelerate the achievable throughput within the job, which will be showcased in Section 5.3.3.4.

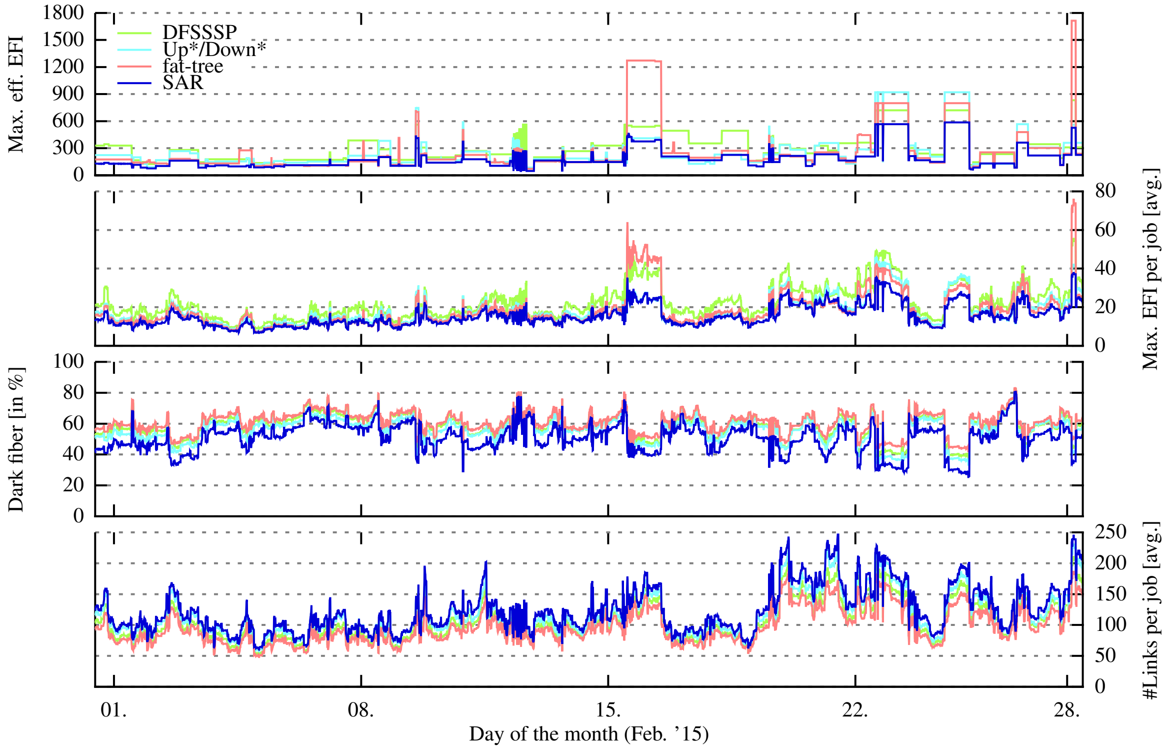
5.3.2.2 Available Links per Job and Dark Fiber Percentage

The edge forwarding index provides upper bounds for worst-case congestion, however a more direct metric for utilization should be evaluated as well. One possibility is to analyze the number of switch-to-switch links that are available to each job and to the overall set of jobs. Similarly, the dark fiber percentage of Definition 5.2 specifies the inverse of this link utilization, and reports how much of the interconnection hardware of the supercomputer is theoretically dispensable.

The dark fiber percentage $\theta_{R,J}$ is shown in the third plot of Figure 5.8a and 5.8b for both supercomputers, and is summarized in the third row of Table 5.1 or Table 5.2, respectively. Especially for TSUBAME2,



(a) Taurus petascale supercomputer



(b) TSUBAME2 petascale supercomputer

Figure 5.8: Replay of job history (Figure 5.1) for two HPC systems, sampled every 10 min; Four routings applied per sampling point; Metrics collected: maximal $\gamma^{e}_{R,J}$ across all links (top), $\gamma^{e}(I,R,j)$ averaged for all jobs (2nd plot), dark fiber percentage (3rd plot), and used links averaged for all jobs (bottom); Lower is better for first three plots; Graph of fat-tree overlaps Up*/Down* in Figure 5.8a due to similar results

Table 5.1: Improvements by the scheduling-aware routing compared to DFSSSP, fat-tree, and Up*/Down* routing for the Taurus HPC system

Metric	DFSSSP		fat-tree		Up*/Down*	
	max. / in %	avg. / in %	max. / %	avg. / %	max. / %	avg. / %
$\max_{c_q \in C^*} \gamma^e_{R,J}(c_q)$	279.0 / 50.8	57.3 / 23.3	18.0 / 21.2	-1.1 / -0.6	18.0 / 21.2	-1.1 / -0.6
$\operatorname{avg}_{j \in J} \gamma^e(I,R,j)$	16.0 / 39.0	4.3 / 23.4	1.6 / 11.4	-0.3 / -2.1	1.6 / 11.4	-0.3 / -2.1
$\theta_{R,J}$ [in %]	9.38	6.03	7.64	3.62	7.64	3.62
$\operatorname{avg}_{j \in J}$ #links(j)	16.5 / 15.0	7.7 / 11.1	14.1 / 13.8	6.6 / 9.4	14.1 / 13.8	6.6 / 9.4

Table 5.2: Improvements by the scheduling-aware routing compared to DFSSSP, fat-tree, and Up*/Down* routing for the TSUBAME2 HPC system

Metric	DFSSSP		fat-tree		Up*/Down*	
	max. / in %	avg. / in %	max. / %	avg. / %	max. / %	avg. / %
$\max_{c_q \in C^*} \gamma^e_{R,J}(c_q)$	321.0 / 61.2	119.4 / 38.5	1186.0 / 71.2	80.5 / 29.7	354.0 / 48.7	69.9 / 26.8
$\operatorname{avg}_{j \in J} \gamma^e(I,R,j)$	18.9 / 46.0	6.7 / 30.1	38.0 / 49.8	2.7 / 14.8	10.6 / 29.9	2.4 / 13.2
$\theta_{R,J}$ [in %]	12.06	7.63	17.74	9.99	9.24	5.51
$\operatorname{avg}_{j \in J}$ #links(j)	49.2 / 26.7	18.3 / 17.4	75.1 / 43.9	26.5 / 27.3	22.3 / 14.9	7.4 / 6.4

the improvements to $\theta_{R,J}$ by the scheduling-aware routing is clearly visible in the plot. SAR increases the number of links available to running batch jobs by up to 17.74% for TSUBAME2 in comparison to the default routing on this system. For both systems, SAR consistently increases $\theta_{\mathrm{SAR},J}$ and thus enables a more efficient resource utilization. Furthermore, the scheduling-aware routing utilizes between 3.6% and 9.9% more links on the month-long average than the other oblivious routing mechanisms.

Lastly, the bottom plots of Figure 5.8 show that SAR increases the number of intra-job links as well. The principle “higher is better” applies to these two plots. Theoretically, this leads to a higher effectively available bandwidth for the applications. Depending on the routing algorithm and HPC system, the SAR approach allows the average multi-switch job to use between 6 and 26 more links.

5.3.2.3 Comprehensive Outlier Analysis of Day 16

The most noticeable difference, visible in Figure 5.8b and Table 5.2, between the scheduling-aware routing and the fat-tree routing are the two outliers on day 16 and day 28. The improvement by SAR,

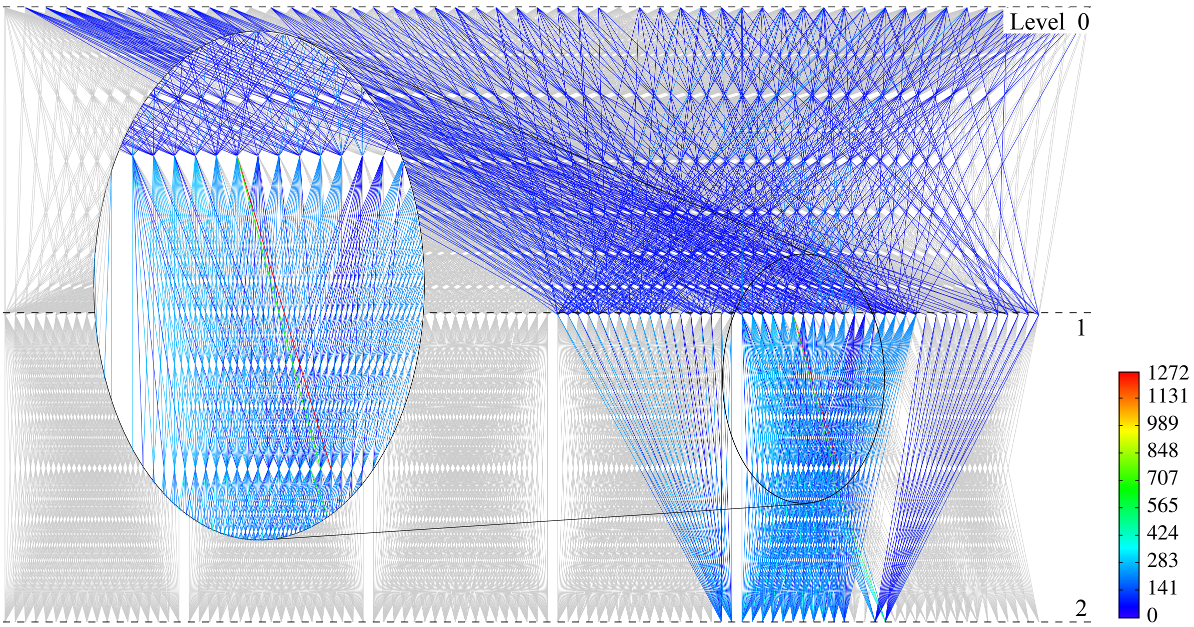


Figure 5.9: Heat map of the EFIs for inter-switch links of TSUBAME2 for the 200-node batch job on day 16 of Feb. '15; Used routing algorithm: fat-tree (Section 3.3.5); Corresponding histogram shown in Figure 5.11a; Link coloring similar to Figure 5.2

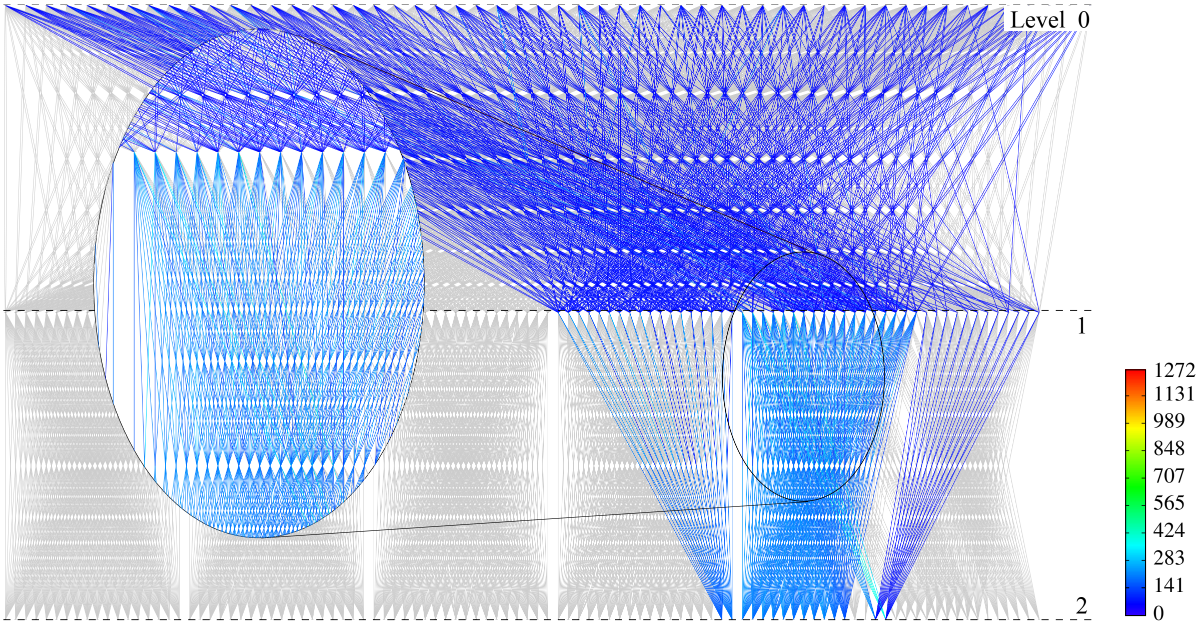
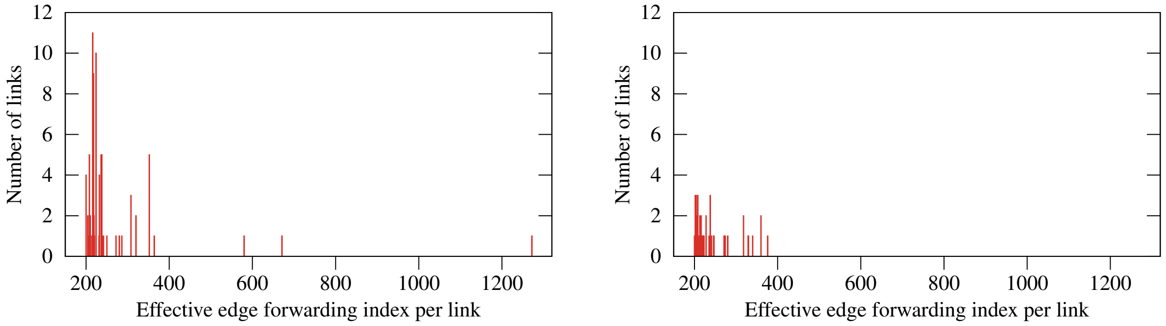


Figure 5.10: Heat map of the effective EFIs $\gamma^{e}_{R,j}$ for inter-switch links of TSUBAME2 for the 200-node batch job j on day 16 of Feb. '15; Used routing algorithm: scheduling-aware routing (Algorithm 5.1); Corresponding histogram shown in Figure 5.11b



(a) Histogram for Figure 5.9

(b) Histogram for Figure 5.10

Figure 5.11: Histograms of effective EFI $\gamma^{e}_{R,j}$ for inter-switch links; Fat-tree routing (left) vs. scheduling-aware routing (right) for the 200-node batch job on TSUBAME2; Limited to $\gamma^{e}_{R,j} \geq 200$

considering all simultaneously executed batch jobs, is up to 72%, which is worth investigating. Analysis of the jobs and effective edge forwarding indices on day 16 of February 2015 shows that one 200-node job is the main reason for the spike. This 200-node scientific application ran for ≈24 h on the TSUBAME2 supercomputer. The nodes belonging to this job were spread across 15 leaf switches of the fabric. The node allocation in combination with the oblivious fat-tree routing yields particularly unfavorable intra-job EFIs, e.g., one port of one level 1 switch forwards a total of 1,272 paths towards the leaf switch, see the red link in Figure 5.9. Gray links indicate network hardware which is unavailable to the batch job. The histogram of links per EFI, for effective edge forwarding indices larger than 200 in Figure 5.11a, reveals two additional ports overloaded with ≈600 intra-job routes. The Figures 5.10 and 5.11b show that SAR is able to reduce $\gamma^{e}(I, \text{SAR}, j)$ for this 200-node job j down to 376. Furthermore, the number of fabric ports available to the 200-node job increases from 1,274 to 1,555 while using SAR, which is not directly visible in Figure 5.11.

5.3.3 Practical Evaluation on a Production System

The scheduling-aware routing, as presented in Section 5.2.2.2, is now deployed for more than one year on the petascale Taurus production system [ZIH13] at the TU Dresden. To recapitulate Section 3.2.6, the system's InfiniBand fabric is designed as a multi-island network. Multiple smaller 2-level full-bisection fat-trees are connected by a 216-port director switch. The following four subsections show numbers collected while the supercomputer was open to regular users. However, although the usual runtime of the SAR implementation in OpenSM is fluctuating around 5 s for the 2,065-node system, detailed runtime comparisons to calculate the linear forwarding tables with SAR are omitted for the following reasons:

- SAR and DFSSSP have the same runtime complexity of $\mathcal{O}\left(|N|^{2} \cdot \log|N|\right)$, for the node set N,
- Measurements revealed a negligible runtime overhead introduced by the SAR extensions, which are shown in Algorithm 5.1, and

- The competitive runtime of DFSSSP routing has been shown before.

For example, refer to related work [DHN11; SBH16] or Section 6.3.3, respectively, for detailed comparisons of the runtime of the deadlock-free SSSP routing.

5.3.3.1 Runtime of the Filtering Tool

The current filtering and reconfiguration tool runs with a 5 min interval, see Figure 5.6, to minimize the chance that SAR reroutes for batch jobs that only run for a very short time. The runtime of the tool including polling the SLURM controller and analyzing currently running jobs has been measured for each run. The average time spent by the filtering tool is 16 s, with a low of 0.02 s. In 99.1% of the cases the runtime was below 2 min. Only three occasions in over a year have been experienced, where the runtime was above 10 min. The vast majority of the runtime of the filtering tool results from the latency to obtain the list of scheduled jobs via the squeue command, which depends on the load of the SLURM controller.

5.3.3.2 New Routing Configurations per Day

The number of reconfigurations of the network per day, due to a substantive change in the job mix and job locations, ranges between 0 and 57, with an average of 14 reconfigurations per day. Therefore, approximately every two hours the system's forwarding tables are adjusted to increase the performance of the running parallel applications. Only for four days in over a year of operation no reconfigurations were needed, and three out of these four days were on weekends, and one time on Monday. Hence, these days match the typical days of lower load on a production system in a university environment.

5.3.3.3 Runtime for Configuring LFTs of all IB Switches

The time it takes to reconfigure all switches in the IB network is of importance. Either because it defines the time window, where out-of-order packet delivery can happen if property preserving network updates are not enforced, or because it defines the time frame during which the network throughput might be decreased due to the use of the highLIDs, which are routed by the less efficient Up*/Down*.

An average of 4.6 μs have been measured to send an LFT block, which contains only 64 egress ports, to a switch and receive an acknowledgment. Hence, the number of blocks per switch is $\lceil \frac{\max(\text{LID})}{64} \rceil$, taking into account that the maximum LID assigned in the system by the subnet manager can be considerably larger than $|N|$. Instead of reconfiguring all LFT blocks of a switch at once, OpenSM iterates over the block numbers and sends out the first LFT block of each switch, before processing the second block, and so on. So, theoretically OpenSM could reconfigure one block number for all 210 switches in the Taurus system in under 1 ms. However, measurement revealed a runtime between 25 ms and 50 ms for this process. Despite this OpenSM-internal overhead, the whole LFT reconfiguration for the supercomputer is completed in $\approx$0.8 s on average. Hence, the likelihood for an out-of-order packet delivery, which cannot be compensated by InfiniBand's packet timeout/retry mechanism (usually configured by Open MPI to be

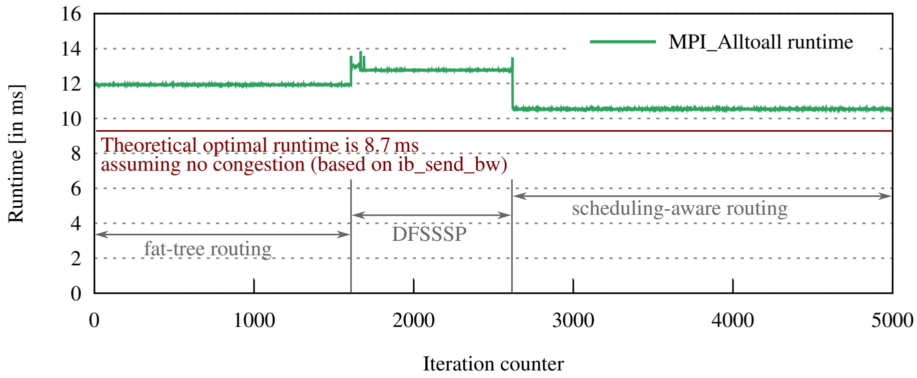


Figure 5.12: Runtime measurement for MPI_Alltoall (with 1 MiByte in each send buffer) on 28 compute nodes using three different routing approaches

larger than 30 s) and which therefore causes an application crash, is close to zero. In fact, no application crashes caused by SAR on the HPC system have been observed, even though the system cannot use the full network update protocol discussed in Section 5.2.3 due to limitations in the HCA firmware.

5.3.3.4 Effects on MPI Traffic

The optimal zero-load latency between any two nodes in the system is not affected by the SAR approach, due to the fact that the underlying algorithm always calculates shortest paths. Therefore, any reconfiguration of the fabric with Algorithm 5.1 will not increase the number of hops between nodes. Changes in the observable latency for small messages are possible, i.e., worse than the optimal latency for individual messages, which depends on the congestion in the system. However, from the two network metrics evaluated in Section 5.3.2, it can deduced that the likelihood of congestion is reduced and therefore the observable latency for small messages improves on average while using SAR. Hence, latency measurements or simulations are not subject of this thesis, since the optimal latency does not change.

Instead, an MPI runtime measurement is conducted on the production HPC system to showcase the impact of the scheduling-aware routing compared to other routing approaches while trying to fully utilizing the available throughput and stress the network. The benchmark emulates a bulk-synchronous application using an MPI_Alltoall collective operation followed by an MPI_Barrier to synchronize the time measurement. Every MPI process sends 1 MiByte to every other process, and the runtime of the slowest process is written into a log file. This benchmark is scheduled to 28 nodes of one full-bisection fat-tree island of Taurus. While the benchmark was running simultaneously with jobs of other users, the routing has been switched between three different routing engines. Due to natural fragmentation of the production system, the 28 nodes ended up connected to ten different switches in the Taurus island.

Figure 5.12 summarizes the results: First, the default fat-tree routing configures the LFTs for the first three minutes. Then, the subnet manager switches to standard DFSSSP routing, approximately between iteration 1,600 and 2,600. The observed runtime increases by 7.1% due to the change from fat-tree routing to DFSSSP routing. For the last time segment, the filtering tool triggers the SAR approach, which initiates

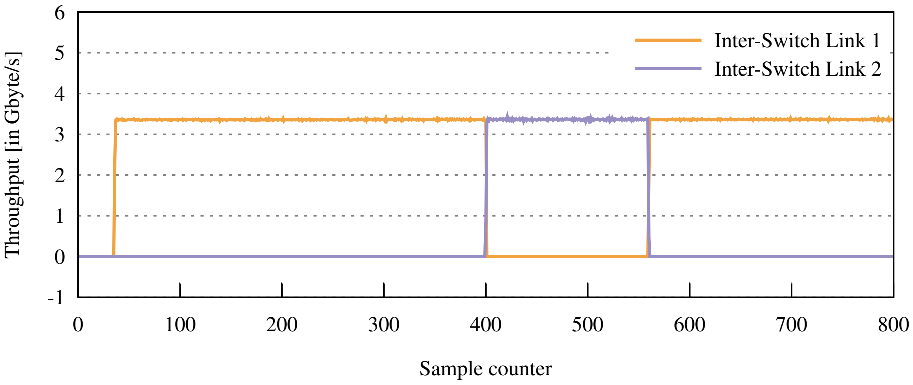


Figure 5.13: Visualization of the network update protocol (without QP draining) and path migration between two inter-switch links on a testbed during high MPI load

the scheduling-aware routing for the whole system. The path optimizations by the scheduling-aware routing decreases the runtime for the benchmark by 17.6% compared to DFSSSP routing and 11.7% compared to fat-tree routing. The plot also includes the theoretical peak performance, assuming all data can be transferred without congestion and MPI library overhead. This shows that SAR essentially halves the congestion overhead caused by the highly optimized fat-tree routing, even on a regular fat-tree.

5.3.3.5 Property Preserving Network Updates on Testbed

Even so draining of the queue pairs currently does not work, i.e., triggering the SQD event as outlined in Section 5.3.1, the OpenSM is able to successfully perform path migrations and update the routing configuration. A small test system is used to showcase that the network updates are safe in practice. This testbed consists of two IB QDR switches and four nodes with IB FDR host channel adapters, with two nodes connected to each switch. Additionally, the two IB switches are connected by two links.

The modified OpenSM, which supports the network update protocol, is running on an administration node. An MPI benchmark, which repeatedly performs MPI_Bcast with a 1 MiByte send buffer, is executed on two compute nodes to emulate a contiguous flow of data from one to the other. Meanwhile, the OpenSM performs a scheduling-aware rerouting, see Figure 5.13. Performance counters of the inter-switch links are queried approximately every 0.07 s while running the benchmark. These counters, in particular PortXmitData [Inf15, 16.1.3.5], are used to calculate the shown throughput. Additionally, OpenSM adds an artificial delay of 10 s between sending unpath and repath traps to highlight the path migration. The area between sample 400 and 560 shows this delay, i.e., phase 2 and 3 of the update protocol, where traffic is routed via the highLID or inter-switch link 2, respectively. The PortXmitData counter of the utilized link is indicating an increment of $\approx$215 Mbyte per sample interval, which adds up to $\approx$3.1 Gbyte/s, due to the 0.07 s sample rate. These values correlate with the maximal throughput of QDR links, i.e., the bottleneck between the two switches. The path/flow migration between the two links is clearly visible in Figure 5.13 and the update protocol has no negative impact on the throughput of the broadcast operation.

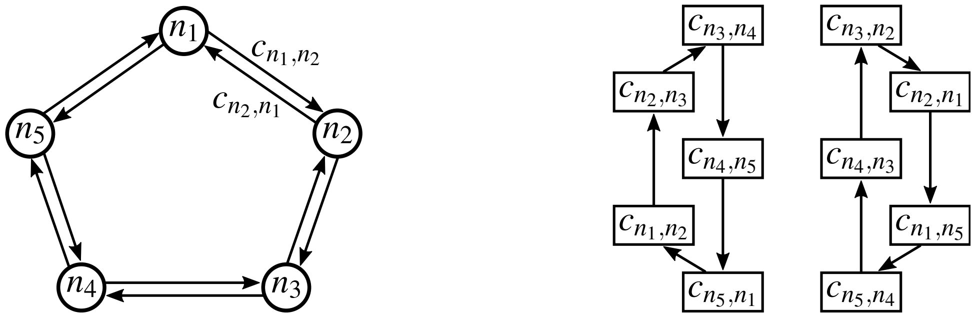
6 Routing on the Channel Dependency Graph

Despite the advances in recent years, to develop new deadlock-free routings for a variety of interconnection technologies and topologies or to improve their efficiency, as introduced in Chapter 5, a major problem persists. No algorithmic approach is known to perform deadlock-free destination-based routing which is generally applicable to all technologies and topologies, and delivers better performance than Up*/Down*, compare to Table 2.1. Such a routing function would be usable on at least 85% of the supercomputers of the current TOP500 list, only counting the systems based on IB, Ethernet, and Omni-Path in Figure 1.1a. This Chapter introduces a novel approach of routing on the channel dependency graph to overcome these issues. Section 6.1 elaborates further on the shortcomings when path finding and deadlock-freedom of the routing algorithm is considered separately. The following Section 6.2 outlines the new routing approach, called Nue, combining the applicability of Up*/Down* with the path balancing of DFSSSP. Nue will be evaluated in Section 6.3 in terms of multiple network metrics, such as path length, path balancing, and achievable throughput, and will be compared to other state-of-the-art routings. Furthermore, the general applicability of Nue and its runtime to calculate a new network state are demonstrated.

6.1 Limitations of Multi-Step Routing Approaches

Two general concepts exist for creating deadlock-free routing functions, both consisting out of two phases, as described hereafter. The first concept is an analytical solution, also referred to as turn model, which basically restricts the usage of possible turns within the network as a first phase, either by knowledge of the underlying topology or similarly to the Up*/Down* routing, before calculating the routes in the second phase. Examples are the topology-aware DOR routing, see Section 3.3.3, L-turn, or multiple Up*/Down* (MUD), listed in Table 2.1. Unfortunately, this approach generally results in non-shortest paths and poor path balancing, and therefore high latency and low throughput, respectively.

In contrast, the current best practice of the second concept, e.g., as implemented by DFSSSP and LASH, see Section 3.3, is to decouple the two problems of path creation into phase 1 and deadlock-free assignment to virtual channels into phase 2. The reason is that both problems require a different graph representation of the network and routes. The deadlock-free assignment needs the channel dependency graph, recall Definition 3.7, an abstract graph induced by the routes, while the route calculation has to take place beforehand and is usually performed on a graph identical to the network. Generally, this approach cannot be bound with respect to the number of virtual channels required to solve the deadlock problem, which is illustrated by the following Proposition 6.1 and network example.



(a) 5-node ring network **(b)** Corresponding cyclic CDG

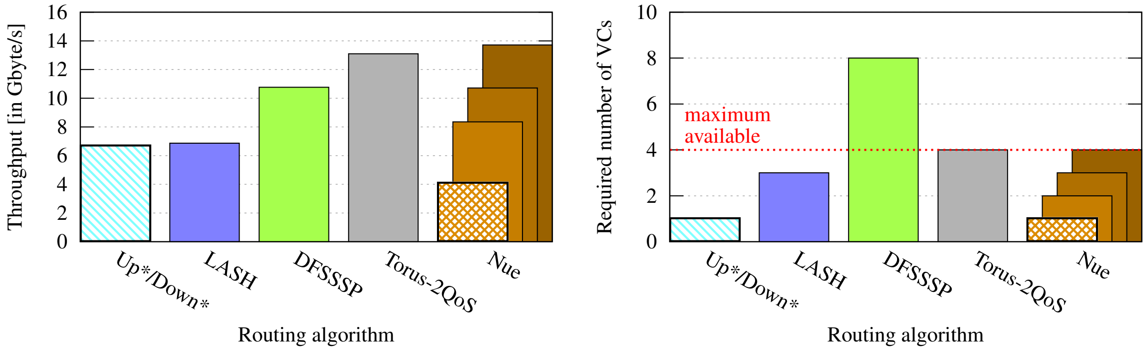
Figure 6.1: Using a shortest-path routing for 5-node ring network I (6.1a) induces a cyclic channel dependency graph D (6.1b) with two cycles

Lemma 6.1. *Let $R : C \times N \to C$ be a given routing function (or more generally $R' : N \times N \to C$), see Definition 3.4, requiring a minimum of $k_{(I,R)} \in \mathbb{N}$ virtual channels for deadlock-freedom on an arbitrary network $I = G(N,C)$. Furthermore, let $k \in \mathbb{N}$ be an arbitrary but limited number of virtual channels, with $k \geq 1$, provided by the interconnection technology. Assuming, R defines solely shortest paths ${}^{s}P_{n_x,n_y}$ for any pair of nodes $n_x, n_y \in N$, then a network topology $I' = G(N,C)$ exists such that $k_{(I',R)} > k$, hence rendering the routing R inapplicable to I'.*

Proof. Let's assume the opposite of the lemma is true, i.e., for any network I and given number of virtual channels $k \in \mathbb{N}$ there exists a deadlock-free and shortest-path routing R with $k_{(I,R)} \leq k$. For example, assuming the absence of virtual channels (equivalent to $k = 1$) for a five-node ring topology, as shown in Figure 6.1a, then a deadlock-free and shortest-path routing should still exist. Any shortest-path routing for this network would route ${}^{s}P_{n_1,n_3}$ via n_3 connecting c_{n_1,n_2} with c_{n_2,n_3}, route ${}^{s}P_{n_2,n_4}$ via n_3 connecting c_{n_2,n_3} with c_{n_3,n_4}, etc. Therefore, the resulting channel dependency graph depicted in Figure 6.1b is the same for any shortest-path routing. Applying Theorem 3.1 reveals that all these routing functions are not deadlock-free, since no virtual channels are available to break the cyclic channel dependency graph. □

Furthermore, determining the minimal number of required virtual channels to provide a deadlock-free path-to-virtual layer assignment is NP-complete [DHN11]. This proposition and the trivial counter example in Figure 6.1 show that a two-phase approach, which first calculates shortest paths through the network for all node pairs, is impractical for designing an universally applicable routing, which allows low latency and high throughput.

Assuming, it is possible to combine the information required to solve both problems within one graph, then this might allow to impose routing restrictions to the path creation on-demand, because the effects of a partial or full path on the CDG can be checked simultaneously. Hence, it would be possible to avoid closing cycles in the CDG, while calculating the paths, instead of breaking the cycles later. Assume, this new graph represents one virtual layer, see Definition 3.8, then a graph search algorithm, such as Dijkstra's algorithm, can traverse the graph and construct routes from all nodes to all other network nodes and the routes are deadlock-free within this layer. The type of graph search and the information assigned



(a) Throughput for an all-to-all pattern

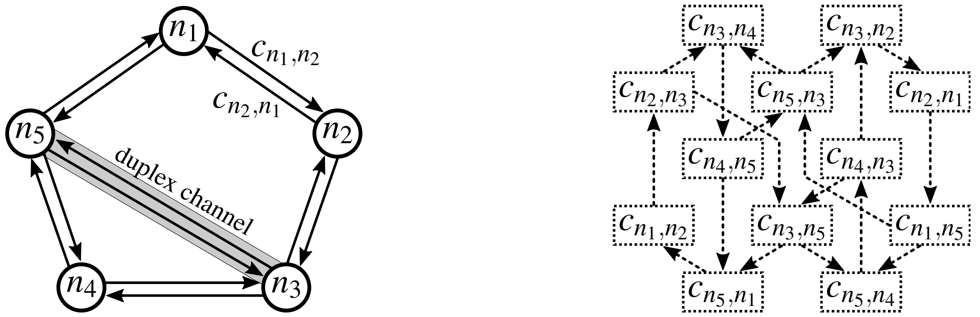
(b) Required VCs for deadlock-freedom

Figure 6.2: Simulated throughput for an all-to-all operation and required VCs for deadlock-freedom for different routing algorithms (hatched bars indicate that only 1 VC is used); Simulated network: 4x4x3 3D torus, 4 terminals per switch, 1 faulty switch, QDR InfiniBand, maximum of 4 VCs available

to this graph influence the resulting routes, e.g., source-routing or destination-based routing could be possible. Furthermore, assuming the used network technology supports an arbitrary, but fixed, number of virtual channels, $k > 1$, then individual destination nodes can be assigned to different virtual layers. As a consequence, the graph search algorithm within one layer is able to calculate deadlock-free routes for all source nodes to the subset of destination nodes assigned to this virtual layer. Therefore, all routes in all virtual layers are deadlock-free without exceeding the virtual channel constraint.

To sketch this idea, a 3-dimensional InfiniBand torus(4,4,3) network with four terminals per switch and one failed switch, i.e., 47 switches in total, is given as an example topology. This IB network shall have network support for four virtual channels, or IB virtual lanes, respectively. The throughput for various deadlock-free routing algorithms, as implemented in InfiniBand's subnet manager, is shown in Figure 6.2a. The shift exchange pattern of Section 4.3.3.2 with 2 KiByte messages yields the given simulation results. Additionally, Figure 6.2b shows the needed number of VCs for these deadlock-free routing algorithms.

The topology-aware Torus-2QoS routing, see Section 3.3.7, enables a high throughput within the virtual channel limit, but will fail if a second switch failure occurs in the same torus ring. Up*/Down* routing and LASH, details in Sections 3.3.4 and 3.3.6, are inefficient in comparison to Torus-2QoS. The throughput of the topology-agnostic routing DFSSSP [DHN11] is in-between, however the deadlock-free single-source shortest-path routing exceeds the given VC limit and is therefore inapplicable. The achievable throughput while using the novel routing approach, which Section 6.2 will elaborate in detail, for the 4x4x3 torus is included in Figure 6.2a for every number of virtual channels within the 4 VC limit. Hence, Nue shows resiliency to network faults and the ability to offer competitive throughput. Surprisingly, Up*/Down* routing slightly outperforms Nue routing in the absence of virtual channels. However, it is noteworthy that the induced CDG of the Up*/Down* implementation in OpenSM actually contains a cycle for this topology and is therefore not deadlock-free, even so the simulated communication pattern did not deadlock. A corrected Up*/Down* implementation probably decreases the achievable throughput even further.



(a) 5-node ring network with shortcut **(b)** Corresponding complete CDG $\overline{D}$

Figure 6.3: Complete CDG $\overline{D}$ for the 5-ring network with shortcut, see Figure 6.3a, assuming $k = 1$; All channels are in the *unused* state (⇒ no routing applied, yet)

6.2 Nue Routing

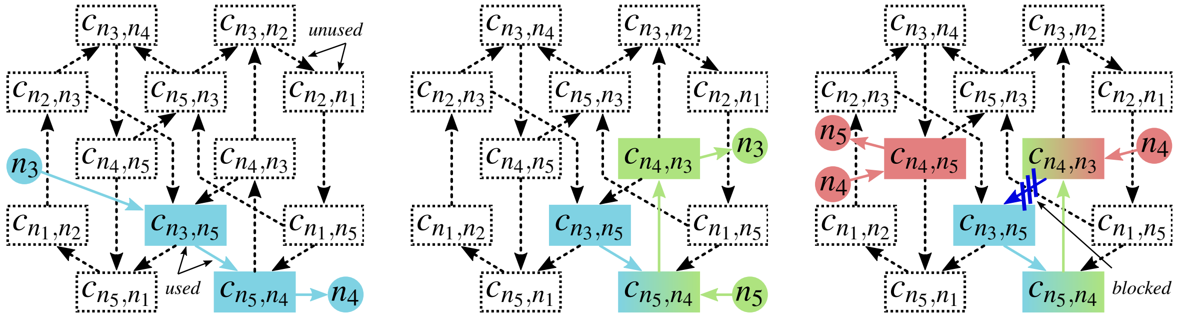
The graph capable of combining the information required to solve the path calculation as well as the virtual channel assignment is called the complete channel dependency graph. The following Section 6.2.1 will outline how to construct the complete channel dependency graph. Based on this graph, a deadlock-free, oblivious, and destination-based Nue routing is developed in Sections 6.2.2 – 6.2.5, which serves as one example for the general idea of routing within the dependency graph. Section 6.2.6 introduces potential improvements for Nue to speedup the path calculation and to increase the achievable throughput on the network. Moreover, Section 6.2.7 proves the correctness and completeness of Nue routing, and analyzes the algorithm's time and memory complexity.

6.2.1 Complete Channel Dependency Graph

To create paths within the CDG, a complete representation of all possible channel dependencies is needed, instead of a graph solely induced by a specific routing algorithm. Therefore, the complete channel dependency graph, using the adjacency of channels, is defined as follows:

Definition 6.1 (Complete Channel Dependency Graph)**.** *Let $I = G(N,C)$ be a network according to Definition 3.1, then the* ***complete channel dependency graph*** $\overline{D} := G(C,\overline{E})$*, with* $\overline{E} \subseteq C \times C$*, is defined by* $\forall (n_x,n_y),(n_y,n_z) \in C, n_x \neq n_z : \big((n_x,n_y),(n_y,n_z)\big) \in \overline{E}$*. Further, the graph* $\overline{D}$ *is defined to be* ***cycle-free****, if* $D \subseteq \overline{D}$ *is acyclic for any CDG D induced by a routing function according to Definition 3.7. Assuming, the network technology supports* $k > 1$ *virtual channels, then the definition of the i-th complete channel dependency graph* $\overline{D}_i := G(C_i,\overline{E}_i)$*, with* $\overline{E}_i \subseteq C_i \times C_i$*, is equivalent to the definition of* $\overline{D}$*.*

The conventional channel dependency graph D induced by a routing function does not have to be stored separately, and can be saved indirectly by assigning states to the vertices, i.e., channels of I and edges of the complete CDG $\overline{D}$. These states are *unused*, *used*, or *blocked*, whereby the *blocked* state is only used for edges. An edge $e \in \overline{E}$ is considered to be *used* if and only if $e \in E$, i.e., e is induced by a routing R. Furthermore, an edge $e \in \overline{E}$ is marked as *blocked* if and only if $G(C, E \cup \{e\})$ forms a cyclic graph for an



(a) Step one: P_{n_3,n_4} via n_5 ✓ **(b)** Step two: P_{n_5,n_3} via n_4 ✓ **(c)** Step three: P_{n_4,n_5} via n_3? ✗

Figure 6.4: Sketching the process of finding paths between network nodes within the complete CDG $\overline{D}$ while avoiding to close any cycles in the induced CDG D

acyclic CDG $D = G(C,E)$. Figure 6.3 shows the complete channel dependency graph $\overline{D}$ for the 5-node ring network with shortcut, shown in Figure 6.3a, for $k = 1$. Each vertex/edge of $\overline{D}$ is in the *unused* state, i.e., no routing has been applied. This ring network is the same as used earlier in Figure 3.1 and subsequent exemplification, and which will be used throughout the remainder of this section.

Finding paths within the complete channel dependency graph $\overline{D}$ is almost as trivial as finding the routes within the original network I, as sketched hereafter. Without going into great detail how the routing algorithm maps paths into the complete CDG and where the routing starts the search, let us assume the algorithm first of all searches a route from network node n_3 to n_4 and finds the possible path P_{n_3,n_4} via n_5, as depicted in Figure 6.4a. Hence, both, the vertices c_{n_3,n_5} and c_{n_5,n_4}, and the edge in between the two, are changed into the *used* state. In step 2, see Figure 6.4b, the routing algorithms tries to determine a route from node n_5 to n_3. A solution is start at c_{n_5,n_4} to traverse the path $P_{n_5,n_3} = (c_{n_5,n_4}, c_{n_4,n_3})$ in the directed graph $\overline{D}$, changing the involved vertices and the edge into the *used* state. In the next iteration of the algorithm, step 3, the route from node n_4 to n_5 should be calculated. If the routing starts at vertex c_{n_4,n_3}, as shown on the right side of Figure 6.4c, then the only possible path to reach n_5 within $\overline{D}$ requires the usage of the edge $(c_{n_4,n_3}, c_{n_3,n_5})$. However, changing this edge into the *used* state is prohibited, since it would close a cycle in the induced CDG D or the *used* marked subgraph of $\overline{D}$, respectively. Hence, the algorithm must assign the *blocked* state to the edge $(c_{n_4,n_3}, c_{n_3,n_5})$, and must search an alternative path by starting at vertex c_{n_4,n_5}, which directly yields a valid path $P_{n_4,n_5} = (c_{n_4,n_5})$.

6.2.2 Escape Paths for the Static Nue Routing

Cherkasova et al. [CKR96] made an important observation during the development of their smart routing algorithm: An incremental algorithm calculating paths and adding routing restrictions at the same time, i.e., prohibiting the assignment $R(c_q, \cdot) = c_{q+1}$ for any destination node, can lead to a routing impasse. Basically, an iteratively working graph algorithm which does not allow backtracking and does not have all-embracing knowledge about the graph might not traverse all nodes of the graph. Hence, further progress in the algorithm, completing the node search, is impossible due to previously added routing restrictions. While Cherkasova et al. [CKR96] report that this state was observed only rarely for their investigated networks, the experiments conducted for this thesis showed that it is a permanent problem for larger networks.

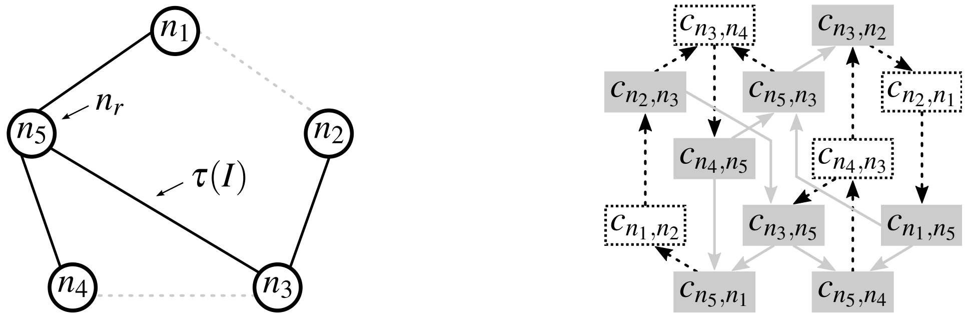
For adaptive routing algorithms it is common to avoid deadlocks by utilizing a separate set of buffers, similar to a virtual layer, which acts as "escape paths" [DT03; Gar+13]. Within this layer a fixed deadlock-free routing, such as Up*/Down*, is employed, and switches transfer a blocked packet into the escape paths for the remainder of the route to its destination.

It is necessary to adapt the concept of escape paths for the oblivious Nue routing to ensure that at least one valid path between every given node pair exists, which does not induce a cycle in the corresponding channel dependency graph. However, these escape paths are not implemented similarly to adaptive routings with a separate set of buffers, but as available fall back paths within a virtual network layer L_i, see Definition 3.8. This fall back path is not necessarily the shortest path available through the graph. The disadvantages of fixed and predefined escape paths are the imposed channel dependencies which cannot act as routing restrictions. Meaning, Nue assigns the *used* state to a subset of vertices and edges of $\overline{D}$, even so these are not necessarily induced by R. The same holds for Up*/Down* routing, which predefines a much larger (and usually unnecessary) number of routing restrictions, i.e., forbidden "down"→"up" turns, which generally results on poor path balancing. Escape paths for Nue routing inevitably serve as potential imaginary paths which also influence the generation and balancing of real paths by the routing function. Therefore, the escape paths should induce as few channel dependencies as possible while minimizing the average path length across the escape paths. The former requirement theoretically increases the degree of freedom the routing function has in placing routing restrictions on-demand, while the latter ensures that the latency, or path length, respectively, is minimized in case an escape path is the only available routing option.

The preferable approach of defining the escape paths in $\overline{D}$ is to use a spanning tree in I, since a spanning tree does not induce a cyclic CDG while minimizing the number of channels required to connect all nodes in I. The following Definition 6.2 details the construction of the escape paths.

Definition 6.2 (Escape Paths)**.** *Let* $N^d \subseteq N$ *be a set of destination nodes for Nue within the network* $I = G(N,C)$ *and let* $\tau(I) = G(N,C^\tau)$*, with* $C^\tau \subseteq C$*, be a spanning tree of network* I *with root node* $n_r \in N$*. Then, the* ***escape paths*** $D^\tau := G(C^\tau, E^\tau)$ *are a subgraph of* $\overline{D}$ *induced by a routing* $R^\tau : N \times N^d \to C^\tau$*. Assuming* $k > 1$*, then* D_i^τ *for virtual layer* L_i *and root* $n_{r,i}$ *is defined equivalently, with the substitution of* C^τ *by* $C_i^\tau \subseteq C_i$*.*

In this context, the term "spanning tree" refers to the originally undirected duplex channels of I, and therefore for any $(n_x, n_y) \in C^\tau$ it holds that $(n_y, n_x) \in C^\tau$, as well. Obviously, there exists always exactly one cycle-free path between any pair of nodes within a spanning tree. Hence, all cycle-free, destination-based routings R^τ induce the same escape paths D^τ or D_i^τ, respectively, for a given spanning tree. Figure 6.5 shows the escape paths marked in the complete channel dependency graph for $N^d = N$ when n_5 is used as the root node for $\tau(I)$.



(a) Spanning tree $\tau(I)$ rooted at n_5 for I of Figure 6.3a **(b)** Complete CDG with marked escape paths D^τ

Figure 6.5: Acyclic escape paths D^τ for the 5-node ring network with shortcut, see Figure 6.3a, are marked as solid boxes/lines within the complete channel dependency graph $\overline{D}$ (6.5b), assuming $k = 1$, destination set $N^d = N$, root node $n_r = n_5$, and spanning tree $\tau(I) = G(N, C \setminus \{(n_1,n_2),(n_2,n_1),(n_3,n_4),(n_4,n_3)\})$, see 6.5a

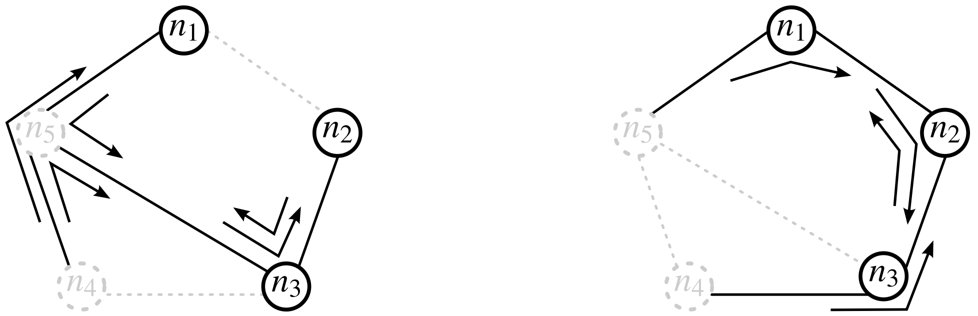
The escape paths for Nue routing have two functions: The first, as mentioned above, is to define an initial set of channel dependencies, which once added to the complete CDG cannot be removed to resolve cyclic states in the routing-induced CDG. Therefore, the escape paths ensure a deadlock-free path from all nodes in N to all nodes in N_i^d within a given virtual layer L_i. Hence, escape paths for each layer are needed, so that at least one valid path exists, which could be found by Nue. Unfortunately, despite of having escape paths D_i^τ, Nue can still end up in an impasse due to the iterative nature of the path creation, which will be shown in Sections 6.2.6.2 and 6.2.6.3. Therefore, secondly and more importantly, the escape paths are to be actively used by Nue after reaching an unsolvable impasse for a destination node, later introduced as the fall back scenario in Section 6.2.6.2.

6.2.3 Choosing Root Node for the Spanning Tree

Nue routing is using escape paths when encountering an impasse, further illustrated in Section 6.2.6.2. Therefore, Nue should ensure that the paths in $\tau(I)$ are as short as possible. This will reduce latency and oversubscription of individual channels with paths in case of a required fall back.

Assuming $N_i^d \subset N$ is a true subset, then an observation is that the number of initial channel dependencies derived from the escape paths depends on the location of the root node as well. The previously investigated 5-node ring topology with shortcut, recall Figure 6.3a, can be examined again to illustrate this property. A first choice for the root node of the spanning tree $\tau(I)$ could be n_5, since it allows for the shortest average path length in the spanning tree. However, if n_5 is chosen as root node, then the escape paths for subset $N_i^d = \{n_1,n_2,n_3\}$, as specified in Definition 6.2, induce five initial channel dependencies, see the five arrows in Figure 6.6a. Alternatively, if $n_r = n_2$ is chosen instead, then only four initial channel dependencies are predetermined, as shown in Figure 6.6b.

As a consequence, a possible approach is to use a root node which is the most central with respect to the subset $N_i^d \subset N$ to reduce the initial channel dependencies within virtual layer L_i. Freeman et al. [Fre77] introduced the metric of betweenness centrality, which is ideal for this purpose to determine the root



(a) Five channel dependencies for $n_r = n_5$ **(b)** Four channel dependencies for $n_r = n_2$

Figure 6.6: Arrows show the initial channel dependencies of the escape paths for the node set $N_i^d = \{n_1, n_2, n_3\}$ and root node n_5 (left) or root node n_2 (right) of the spanning tree marked in black

node for the spanning tree. For a given graph $G(N,C)$, the betweenness centrality $C_B(n)$ for a node $n \in N$ is defined through the ratio of all shortest paths to shortest paths crossing n, see Equation 6.1 hereafter. Let $\sigma_{uv} := \left|\{{}^sP_{u,v} \mid u,v \in N\}\right|$ be the absolute number of distinguishable shortest paths between any two nodes u and v, and let $\sigma_{uv}(n) := \left|\{{}^sP_{u,v} \mid u,v \in N \wedge (\cdot,n) \in {}^sP_{u,v}\}\right|$ be the number of shortest paths which include node n, then the betweenness centrality is equal to:

$$C_B(n) := \sum_{n \neq u \neq v \in N} \frac{\sigma_{uv}(n)}{\sigma_{uv}} \tag{6.1}$$

Brandes et al. [Bra01] developed an algorithm to compute the betweenness centrality $C_B(n)$ of every node $n \in N$ in the graph, to determine the most central node with respect to N. The algorithm for unweighted graphs has an $\mathcal{O}(|N| \cdot |C|)$ time complexity. Unfortunately, the algorithm is not directly applicable to the previously stated problem (except for the trivial case of $N^d = N$), because the most central node with respect to the subset $N_i^d \subset N$ is required to reduce the initial channel dependencies of the escape paths towards nodes of N_i^d. The proposed alternative for Nue is to solve a related problem, i.e., to calculate the convex subgraph for the node set N_i^d, and apply Brandes' algorithm on the convex subgraph. The following Definition 6.3 specifies the construction of the convex subgraph for N_i^d:

Definition 6.3 (Convex Subgraph)**.** *The* ***convex subgraph*** $H_i := G(N_i^H, C_i^H)$ *for a set* $N_i^d \subseteq N$ *includes all nodes of* N_i^d*, as well as some nodes of* $N \setminus N_i^d$ *which are intermediate nodes of the shortest paths* ${}^sP_{\cdot,\cdot}$ *between nodes of* N_i^d*. Therefore, the node set of* H_i *is defined as*

$$N_i^H := \{n \in N \mid n = n_x \vee \exists n_x, n_y \in N_i^d : (\cdot,n) \in {}^sP_{n_x,n_y}\}$$

and the edge set C_i^H *of the convex subgraph is defined analogously.*

In this thesis, interconnection networks are represented by an unweighted graph, see Definition 3.1, which allows the computation of the convex subgraph with a time complexity of $\mathcal{O}\big(|N_i^d| \cdot (|N| + |C|)\big)$ with

Algorithm 6.1: Convex subgraph for node set N^d of an unweighted graph $G(N,C)$

```
   Input: $I = G(N,C)$, node subset $N^d \subseteq N$
   Result: Node set $N^H$ of the convex subgraph $H$ for $N^d$
1  $N^H \leftarrow N^d$
2  foreach node $n_x \in N^d$ do
3      foreach node $n \in N$ do
4          $n$.distance $\leftarrow \infty$
5          $n$.processed $\leftarrow$ false
       /* forward step                                          */
6      FIFO queue $Q \leftarrow \{n_x\}$
7      while $Q \neq \emptyset$ do
8          $u \leftarrow Q$.dequeue()
9          foreach $(u,v) \in C$ do
10             if $v.distance = \infty$ then
11                 $v$.distance $\leftarrow u$.distance $+ 1$
12                 $Q$.enqueue($v$)
       /* backward step                                         */
13     foreach node $n_y \in N^d$ do
14         $n_y$.processed $\leftarrow$ true
15         FIFO queue $Q \leftarrow \{n_y\}$
16         while $Q \neq \emptyset$ do
17             $u \leftarrow Q$.dequeue()
18             foreach $(u,v) \in C$ with $v.processed =$ false do
19                 if $v.distance = u.distance - 1$ then
20                     if $v \notin N^H$ then $N^H \leftarrow \{v\} \cup N^H$
21                     $v$.processed $\leftarrow$ true
22                     $Q$.enqueue($v$)
```

a breadth-first search (forward step) and an inverse traversal of the graph to find all shortest paths (backward step), as shown in Algorithm 6.1. This algorithm is derived from Dijkstra's algorithm to compute the shortest paths from one node to all other nodes. In the forward step, the distances from $n_x \in N_i^d$ to all nodes in N are computed. The subsequent backward step then traverses from all $n_y \in N_i^d$ towards n_x and adds all nodes to the node set of H_i, which have not been processed before. The break at already processed nodes, see loop condition at line 18, is possible, because it means that all shortest paths from this node towards n_x have been identified earlier. The depicted pseudocode omits the layer index i, because the algorithm is identical for each virtual layer and the only difference between layers is the node set N_i^d. Section 6.2.5 elaborates the creation of these subsets of N in Algorithm 6.3.

After computing the convex subgraph H_i, Brandes' algorithm is executed on H_i instead of I to find $n_{r,i} \in N_i^H$ which maximizes the betweenness centrality $C_B(n)$ with respect to N_i^d. This node is used as the root node of the spanning tree for the escape paths from all nodes towards the destination nodes in N_i^d, to define the initial (non-removable) dependencies, see Section 6.2.2. Note that, for $k = 1$, Nue assigns all destination nodes N_i^d to one virtual layer. Hence, H_1 is equivalent to the network I and Brandes' algorithm can be executed directly on I.

6.2.4 Dijkstra's Algorithm for the complete CDG

Previous research [HSL09; DHN11], as well as the introduced scheduling-aware routing optimization in Chapter 5, shows the effectiveness of a routing algorithm based on Dijkstra's single-source shortest-path

Algorithm 6.2: Modified Dijkstra's algorithm within the complete CDG $\overline{D}$

```
Input: $I = G(N,C)$, $\overline{D} = G(C,\overline{E})$, source $n_0 \in N^d$
Result: $P_{n_y,n_0}$ for all $n_y \in N$ (and $\overline{D}$ is cycle-free)
foreach node $n \in N$ do
    $n$.distance $\leftarrow \infty$
    $n$.usedChannel $\leftarrow \emptyset$
foreach channel $c \in C$ do
    $c$.distance $\leftarrow \infty$
/* Need a source channel $c_0$ to start the Dijkstra's algorithm:  if $n_0$ is terminal then use unique $(n_0,\cdot)$;
   if $n_0$ is switch then $\overline{D}$ has multiple $(n_0,\cdot)$ => use fake channel $(\emptyset,n_0)$ to connect to all      */
if $n_0$ is switch then $c_0 \leftarrow (\emptyset,n_0)$ else $c_0 \leftarrow (n_0,\cdot)$
$n_0$.distance $\leftarrow 0$
$c_0$.distance $\leftarrow 0$
FibonacciHeap $Q \leftarrow \{c_0\}$
while $Q \neq \emptyset$ do
    $c_p \leftarrow Q$.findMin()
    foreach $(c_p,c_q) \in \overline{E}$ with $(c_p,c_q)$.state $\neq$ blocked do
        // Let $n_{c_q} \in N$ be the tail of directed channel $c_q$
        if $c_p$.distance $+ c_q$.weight $< n_{c_q}$.distance then
            $(c_p,c_q)$.state $\leftarrow$ used                                  // modifies $\overline{D}$
            if $\overline{D}$ is cycle-free then              // cycle check is outlined in Section 6.2.6.1
                if $n_{c_q}$.usedChannel $\neq \emptyset$ then
                    $Q$.remove($n_{c_q}$.usedChannel)
                $Q$.add($c_q$)
                $c_q$.distance $\leftarrow c_p$.distance $+ c_q$.weight
                $n_{c_q}$.distance $\leftarrow c_p$.distance $+ c_q$.weight
                $n_{c_q}$.usedChannel $\leftarrow c_q$
            else
                $(c_p,c_q)$.state $\leftarrow$ blocked
        ...
    // Optimizations are explained in Sections 6.2.6.2 and 6.2.6.3
        ...
if $n_0$ is switch then
    remove fake channel $c_0$ from $\overline{D}$ and $(c_0,(n_0,\cdot))$ from $\overline{E}$
```

algorithm. The effectiveness is quantified by the algorithm's ability of balancing the paths across the available channels. The modified Dijkstra algorithm of SSSP, extended to work on multigraphs, see Section 3.3.2, has a low time complexity of $\mathcal{O}\big(|N|^2 \cdot \log|N|\big)$ when used with a Fibonacci heap. The destination-based (DF-)SSSP routing initializes the network channels with positive weights of $|N|^2$ to ensure the calculation of shortest paths. Subsequent path calculations from each network node to all other nodes by Dijkstra's algorithm, executed on the network graph $I = G(N,C)$, define the routes in inverse direction, and therefore the entire network state. The balancing is achieved by updating the channel weights of the used channels after all paths towards one destination node, i.e., the source node for the modified Dijkstra algorithm, are computed.

Nue routing follows a similar approach, but within the complete channel dependency graph $\overline{D}$ or $\overline{D}_i$, respectively, instead of the actual graph representing the network $I = G(N,C)$. Hence, Dijkstra's algorithm needs to be modified to be applicable to $\overline{D}_i$ and enhanced to satisfy additional constraints. The pseudocode shown in Algorithm 6.2 outlines the modified version for the complete CDG $\overline{D}$ only, but it can easily be extended for $\overline{D}_i$ due to their similarity to $\overline{D}$, see Definition 6.1. Algorithm 6.2 computes shortest paths from one source node $n_0 \in N^d$ to all other nodes in the complete CDG while complying to the cycle-free

Algorithm 6.3: Nue routing calculates all paths within a network $I = G(N,C)$ for a given number of virtual channels $k \geq 1$

```
Input: $I = G(N,C), k \in \mathbb{N}$
Result: Path $P_{n_x,n_y}$ for all $n_x, n_y \in N$
Partition $N$ into $k$ disjoint subsets $N_1^d, \ldots, N_k^d$ of destinations
foreach virtual layer $L_i$ with $i \in \{1,\ldots,k\}$ do
    // Check attached comments for details about each step
    Create a convex subgraph $H_i$ for $N_i^d$                              // Section 6.2.3
    Identify central $n_{r,i} \in N_i^H$ of $H_i$ via Brandes' algorithm     // Section 6.2.3
    Create a new complete channel dependency graph $\overline{D}_i$          // Section 6.2.1
    Define escape paths $D_i^{\tau}$ for spanning tree root $n_{r,i}$       // Section 6.2.2
    foreach node $n \in N_i^d$ do
        Identify deadlock-free paths $P_{\cdot,n}$                          // Section 6.2.4
        Store these paths, e.g., in forwarding tables
        Update channel weights in $\overline{D}_i$ for these paths
```

constraint. However, due to this cycle-free constraint, these paths are not necessarily shortest paths with respect to the actual network I. Following these paths in opposite direction along the used channels, i.e., nodes of $\overline{D}$, results in the paths for the destination-based routing. Nue routing initializes, with $|N|^2$, and updates the channel weights similar to DFSSSP [DHN11]. However, the fact that channels are the vertices of $\overline{D}$ changes the computation, see line 13, and weights are stored at the adjacent channel instead of the edge between two channels. The advantage of the approach of routing on the dependency graph is that channel dependencies are directly considered, see line 15, and routing restrictions can be identified instantaneously, as outlined in Section 6.1. Therefore, paths do not have to be recomputed to avoid the routing restriction afterwards, as it is the case with smart routing [CKR96], for example.

6.2.5 Nue Routing Function

Combining the knowledge of Section 6.2.1 – 6.2.4 into one routing function, see Algorithm 6.3, allows Nue to achieve its objectives, i.e., to be able to balance the paths globally while not exceeding a given number of virtual channels, $k \geq 1$, used for deadlock-freedom. In the first step, Nue routing partitions the nodes of the network I into k disjoint subsets, i.e., $N_1^d, \ldots, N_k^d$ with $N_i^d \cap N_j^d = \emptyset$ for all $1 \leq i, j \leq k$, $i \neq j$. Each subset N_i^d will denote a set of destinations for calculated paths, i.e., $P_{\cdot,n}$ for $n \in N_i^d$, within virtual layer L_i. While the exact partitioning of N will not influence whether Nue can calculate deadlock-free routes for I or not, the partitioning affects the path balancing. Nue routing uses a multilevel k-way partitioning algorithm [KK98] with $\mathcal{O}(|C|)$ time complexity to partition the network I. Moreover, random partitioning and partial clustering, i.e., all terminals connected to a switch are assigned to the same partition, have been tested for this thesis. However, Nue with the multilevel k-way partitioning outperformed the other two partitioning algorithms with respect to the evaluations carried out in Section 6.3. An optimal partitioning algorithm, i.e., a partitioning which results in a maximized path balancing and which minimizes the edge forwarding indices for the switches, is beyond the scope of the thesis and requires further research.

The node set N_i^d is used to calculate a convex subgraph H_i. Brandes' algorithm is then executed on H_i to determine the betweenness centrality for each node of H_i ensuring the selection of an appropriate root node $n_{r,i}$ for the escape paths, see Section 6.2.3. After creating a complete CDG $\overline{D}_i$ for virtual layer L_i, which complies to Definition 6.1, Nue routing determines the escape paths D_i^τ. The acyclic escape paths are derived from a spanning tree rooted at $n_{r,i}$ according to Definition 6.2, i.e., the channels C_i^τ and edges E_i^τ are changed into the *used* state. This completes the initialization phase of the complete CDG $\overline{D}_i$ to perform the graph search algorithm within $\overline{D}_i$ with the modified Dijkstra algorithm, see Algorithm 6.2. Each node of N_i^d is used as a source for Algorithm 6.2. The subsequent weight update for the used channels aims for an improved global balancing of the paths.

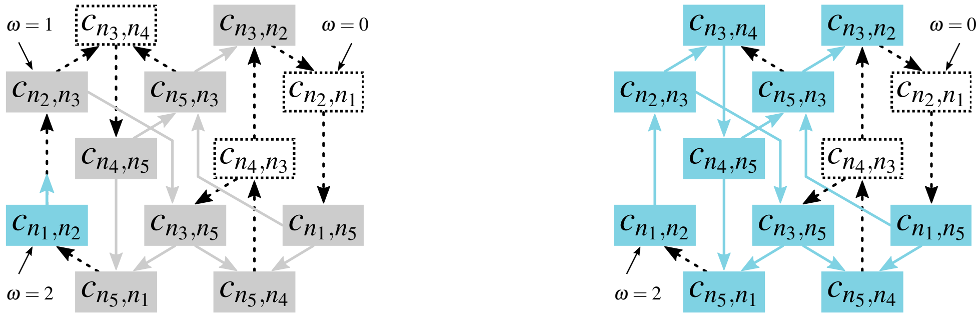
6.2.6 Optimizations for Nue Routing

The modified version of Dijkstra's algorithm, applicable to the complete channel dependency graph, includes a cyclicality check for the graph $\overline{D}$, which has not been looked at so far. Usually, verifying whether or not a graph contains a cycle requires a rather time consuming depth-first search. The time complexity of a depth-first search executed on a graph $G(V,E)$ is $\mathcal{O}(|V|+|E|)$ for the cycle search [Cor+01, Section 22.3]. Therefore, the following Section 6.2.6.1 outlines a possible approach to minimize the time needed to determine the cyclicality of $\overline{D}$. Furthermore, as discussed in Section 6.2.2, Nue routing encounters routing impasses regularly for larger networks. Solving each impass by falling back to the escape paths would yield a path balancing similarly poor as Up*/Down* routing. Hence, Sections 6.2.6.2 and 6.2.6.3 introduce methods to reduce the number of fall backs.

6.2.6.1 Numbering of Subgraphs and Cycle Search

Algorithm 6.2 has an $\mathcal{O}(|C_i| \cdot \log|C_i| + |\overline{E}_i|)$ time complexity, if applied on virtual layer L_i, and if the search for cycles in $\overline{D}_i$ is omitted. This time complexity is derived from Dijkstra's original shortest-path algorithm when used in combination with a Fibonacci heap [FT87]. However, Algorithm 6.2 potentially needs to check the graph $\overline{D}_i$ for cycles every time the state of an edge $(c_p, c_q) = e \in \overline{E}_i$ changes, see line 15. The time complexity of each full cycle search in $\overline{D}_i = G(C_i, \overline{E}_i)$ is $\mathcal{O}(|C_i| + |\overline{E}_i|)$, according to the previous section.

A trivial proposition for directed graphs is that connecting two acyclic digraphs by one directed edge does not create a cyclic digraph. Hence, assuming, it is possible to distinguish between vertex-disjoint, *used* subgraphs of $\overline{D}_i$, induced by a routing R as explained in Section 6.2.1, then cycle searches can be avoided by applying memorization, since connecting two disjoint, acyclic, and *used* subgraphs with an *used* edge creates a new acyclic subgraph. Therefore, Nue routing incorporates an identification number ω for *used* and cycle-free subgraphs of $\overline{D}_i$, which is an extension of the three states which have been utilized



(a) Initial state in Algorithm 6.2

(b) State after five steps

Figure 6.7: State change of $\overline{D}$ after five steps with Algorithm 6.2, starting from c_{n_1,n_2}; Figure 6.7a: initial state (line 9) before the while loop is executed; Figure 6.7b: state of $\overline{D}$ after five iterations of the loop

before, see Section 6.2.1. The function $\omega : C_i \cup \overline{E}_i \rightarrow \mathbb{Z}_0^+ \cup \{-1\}$, with

$$\omega(x) = \begin{cases} -1 & \text{if } D_i \cup x \text{ forms cycle in } \overline{D}_i\text{, i.e., } x \text{ is } \textit{blocked}, \\ 0 & \text{if } x \notin D_i \land x \notin D_i^\tau\text{, i.e., } x \text{ is } \textit{unused}, \\ \geq 1 & \text{if } x \text{ is in the } \textit{used} \text{ state} \end{cases} \tag{6.2}$$

is used to identify the vertex-disjoint, cycle-free subgraphs and *blocked* edges, i.e., $\omega(e) = -1$. An example is shown in Figure 6.7a with $\omega = 1$ pointing to the escape paths of the complete channel dependency graph $\overline{D}$, i.e., $\omega(C^\tau \cup E^\tau) = 1$ for $D^\tau = G(C^\tau, E^\tau)$ assuming $k = 1$.

The advantage of this is to identify conditions during the routing with the modified version of Dijkstra's algorithm where a cycle search is needed or can be omitted. Hence, at node $c_p \in C$ of the complete CDG with assigned $\omega(c_p) \geq 1$ and adjacent node $c_q \in C$, with $(c_p, c_q) =: e \in \overline{E}$, there are four possible conditions. Fortunately, three of these conditions do not require a cycle search:

(i) $\omega(e) = -1 \implies$ no cycle search is needed, because the result is known already, i.e., these edges are ignored by the conditional loop in line 12 of Algorithm 6.2;

(ii) $\omega(e) \geq 1 \implies \omega(c_p) = \omega(c_q) = \omega(e) \implies$ no cycle search is needed, because e was used before and is therefore part of an acyclic subgraph;

(iii) $\omega(e) = 0 \land \omega(c_p) \neq \omega(c_q) \implies$ no cycle search is needed, because the directed edge e connects two disjoint acyclic subgraphs and therefore cannot close a cycle;

(iv) $\omega(e) = 0 \land \omega(c_p) = \omega(c_q) \implies$ a cycle search is required, because e adds an used edge connecting two nodes of a single acyclic subgraph and might induce a cycle.

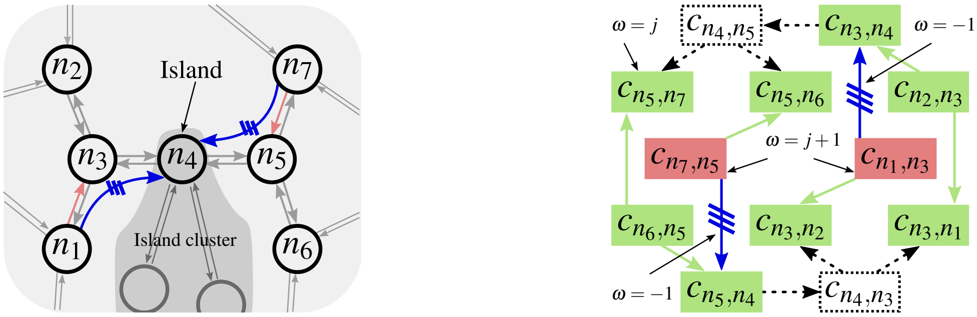
Algorithm 6.4 shows the handling of these conditions, inclusive the performed cycle search with a depth-first search (DFS). For simplicity, the algorithm shows the procedure for $\overline{D}$, i.e., $k = 1$. The

Algorithm 6.4: Search for cyclic *used* subgraphs in the complete CDG $\overline{D}$

```
   Input: $\overline{D} = G(C,\overline{E})$, channels $c_p, c_q \in C$
   Result: true if a cycle was found; false otherwise
 1 if $\omega(c_p,c_q) = -1$ then
 2     return true                                                  // State described in condition (i)
 3 else if $\omega(c_p,c_q) \geq 1$ then
 4     return false                                                 // State described in condition (ii)
 5 else if $\omega(c_p) \neq \omega(c_q)$ then
       /* Merge two disjoint subgraphs                                                                  */
 6     if $\omega(c_q) = 0$ then $\omega(c_q) \leftarrow \omega(c_p)$ and return false
 7     foreach $x \in C \cup \overline{E}$, with $\omega(x) = \omega(c_q)$ do
 8         $\omega(x) \leftarrow \omega(c_p)$
 9     return false                                                 // State described in condition (iii)
10 else
       /* Perform a DFS for $c_p$ in subgraph $G$ with $\omega(G) = \omega(c_p)$ starting from $c_q$, see condition (iv)   */
11     if the $\overline{D}.DFS(c_q, \omega(c_p))$ does not find $c_p$ then
12         $\omega(c_p,c_q) \leftarrow \omega(c_p)$
13         foreach $x \in C \cup \overline{E}$, with $\omega(x) = \omega(c_q) \neq 0$ do
14             $\omega(x) \leftarrow \omega(c_p)$
15         $\omega(c_q) \leftarrow \omega(c_p)$
16         return false
17     else
18         $\omega(c_p,c_q) \leftarrow -1$
19         return true
```

depth-first search is only performed within a selected subgraph of $\overline{D}$ identified by $\omega(c_p)$. Since this subgraph is acyclic without (c_p, c_q), this edge must be part of a new cycle if one exists. Therefore, one depth-first search starting from c_q and searching for c_p is sufficient. Hence, Nue potentially omits to traverse parts of the subgraph, which leads to a more efficient algorithm.

Let us illustrate the conditions (ii) to (iv) with the previously investigated example of the ring topology with shortcut for $k = 1$. Initially, the first cycle-free subgraph is identified by assigning $\omega = 1$ to the escape paths, see light gray vertices and edges of $\overline{D}$ in Figure 6.7a. For the sake of simplicity, the assumption is that the first routing step of Algorithm 6.2 starts at vertex c_{n_1,n_2} (instead of $c_{\emptyset,n_1}$ for source node n_1) and assigns $\omega(c_{n_1,n_2}) = 2$ to it. This will identify the second *used* and cycle-free subgraph of $\overline{D}$, as shown in Figure 6.7a with the dark gray vertex. At this stage, the algorithm has found two nodes of I, the source n_1 and network node n_2. Vertex c_{n_1,n_2} has only one adjacent vertex c_{n_2,n_3} available via an *unused* directed edge. Due to that fact that $\omega(c_{n_1,n_2}) \neq \omega(c_{n_2,n_3})$ holds, a cycle search can be omitted, see condition (iii). According to lines $5-9$ of Algorithm 6.4, both subgraphs, with $\omega = 1$ and $\omega = 2$, are merged into one acyclic subgraph with $\omega = 2$. Now, the adjacent vertices of c_{n_2,n_3} are c_{n_3,n_5} and c_{n_3,n_4} whereby the conditions (ii) and (iii) apply, respectively. Meaning, no cycle search is required for the second routing step, i.e., for both iteration of the loop in line 12 of Algorithm 6.2. Assuming, this algorithm considers vertex c_{n_3,n_4} next, then the only available adjacent vertex is c_{n_4,n_5}, which results in condition (iv), where a depth-first search is needed. A depth-first search, starting from c_{n_4,n_5} and searching for node c_{n_3,n_4} within the subgraph, checks a total of three nodes, i.e., c_{n_5,n_1}, c_{n_5,n_3}, and c_{n_3,n_2}. Since the starting node is not found, it is possible to use $(c_{n_3,n_4}, c_{n_4,n_5})$ without closing a cycle, which concludes the third iteration of the while loop. Two more vertices of Q need to be processed, and the intermediate state of $\overline{D}$ during



(a) Subnetwork I^* of a larger network I

(b) Intermediate state of the complete CDG $\overline{D}$

Figure 6.8: Impasse of Algorithm 6.2 to reach n_4 based on previously placed routing restrictions for channels dependencies $(c_{n_1,n_3}, c_{n_3,n_4})$ and $(c_{n_7,n_5}, c_{n_5,n_4})$, which are shown as crossed out edges in I^* and $\overline{D}$

Algorithm 6.2, after these steps have been performed, is shown in Figure 6.7b. All subsequent executions of Algorithm 6.2 for other network nodes will increase ω of the source vertex by $+1$, hence the next source c_0 gets $\omega(c_0) = 3$ assigned to it, and so forth.

6.2.6.2 Solving Impasses for Isolated Nodes and Node Clusters

The approach of randomly removing channel dependencies, as explained in Section 6.2.2 and mentioned by Cherkasova et al. [CKR96], can lead to impasses during an iterative routing algorithm. Incrementally calculating routes and placing routing restrictions on-demand, as done by Nue routing, can lead to similar impasses. Meaning, creating isolated parts of the network, called *islands* from here on, where no path can be assigned to without creating a cycle in the CDG, based on previously calculated routes for other destinations. Even the escape paths, as introduced in Section 6.2.2, for the iterative, destination-based Nue routing function cannot prevent impasses, but the escape paths offer an expedient for Nue routing.

To illustrate the problem, the previously used example network needs to be substituted with a larger network I, with a small subnetwork I^* connected as a binary tree, as shown in Figure 6.8a, supporting no VCs (i.e., $k = 1$). The subgraph of the complete channel dependency graph $\overline{D}$ for the relevant parts of the network, i.e., for I^*, is shown in Figure 6.8b. Assuming, the iterative Algorithm 6.3 has calculated all routes for $j-1$ destinations, and is at an intermediate step to calculate the routes towards the j^{th} destination in the for loop (line 7). Therefore, parts of $\overline{D}$ will have $\omega = j$ assigned to it, i.e., $\omega = 1$ for the escape paths plus $j-1$ previously processed destinations. Now, Algorithm 6.2 reaches n_3 and n_5 on the shortest path via the channels c_{n_1,n_3} and c_{n_7,n_5}, respectively. Due to previous routing decisions, the channel dependencies $(c_{n_1,n_3}, c_{n_3,n_4})$ and $(c_{n_7,n_5}, c_{n_5,n_4})$ are in the *blocked* state, as illustrated by the crossed out edges. Hence, the routing algorithm reached an impasse and cannot calculate valid routes for node n_4.

There are multiple options to solve this impasse, and three of them are listed hereafter. The easiest among the possible solutions for the impasse is to simply fall back to the escape paths for the entire routing step, i.e., all routes to one specific destination node will use the escape paths instead of the paths

Algorithm 6.5: Local backtracking finds paths into islands

```
   Input: $I = G(N,C), \overline{D} = G(C,\overline{E})$
   Result: Adds one $c \in C$ to FibonacciHeap $Q$ or resets routing to escape paths
   // This is the expansion of Algorithm 6.2 (while loop in line 10)
24     if $Q$ is empty $\wedge \exists n_x \in N : n_x$.usedChannel $= \emptyset$ then
25         foreach $n_x \in N$ with $n_x$.usedChannel $= \emptyset$ do
26             List Alternatives $\leftarrow \emptyset$
27             foreach $n_y \in N$ with $(n_y, n_x) \in C$ do
28                 foreach $c_p \in$ usedChannelStack$(n_y)$ do
                       /* Check, if all existing dependencies on $n_y$ are valid if $c_p$ is used    */
29                     if $(c_p, \cdot)$.state is not blocked then
30                         AlternativeChannels.add($c_p$)
31             AlternativeChannels.sortAscendingDistance()
32             foreach $c_p \in$ AlternativeChannels do
33                 $c_q \leftarrow (n_{c_p}, n_x)$                                  // i.e., $n_{c_q} = n_x$
34                 $(c_p, c_q)$.state $\leftarrow$ used
35                 if $\overline{D}$ is cycle-free then
36                     $Q$.add($c_q$)
37                     $c_q$.distance $\leftarrow c_p$.distance $+ c_q$.weight
38                     $n_x$.distance $\leftarrow c_p$.distance $+ c_q$.weight
39                     $n_x$.usedChannel $\leftarrow c_q$
40                     Island.remove($n_x$)
41                     break                                          // break outer loop, line 25
42                 else
43                     Reset $(c_p, c_q)$.state
44         if $Q$ is empty then
45             Fall back to escape paths, i.e., all paths towards the input node $n_0$ of Algorithm 6.2 use the escape paths
46             Terminate Algorithm 6.2
```

calculated by Algorithm 6.2. The design of the escape paths, recall Definition 6.2, enables this for all destinations N_i^d for the currently considered virtual layer L_i. Unfortunately, falling back to the escape path for only a subset of the paths to one destination can violate the destination-based property of Nue. Hence, this first option is the least preferred, because impasses happen regularly, and the escape paths would be overloaded with routes.

Another option is a backtracking algorithm starting from the current intermediate state of Nue routing and revert previous decisions about chosen paths. However, this means that potentially all previous chosen (partial) paths to the current destination have to be changed, due to the channel dependencies. This results in a brute-force algorithm, because the algorithm has no knowledge which "wrong decision" in the beginning leads to the impasse. The method would guarantee a solution, since at least one valid solution, i.e., the escape paths, exists, but it greatly increases the runtime, rendering the algorithm impractical.

The proposed approach for Nue routing is to use a local backtracking algorithm, whereby only the surrounding nodes in a distance of two hops are checked for alternative routes to the island. This can be accomplished both time- and memory-efficient, see Algorithm 6.5. If no alternative path can be found, which happens less frequent, then Nue falls back to the escape paths as described in the first option, see lines 44 – 46 of Algorithm 6.5. So, instead of having Algorithm 6.2 to overwrite the used channel, see line 21, the used channels are stored in a stack. Hence, this stack of valid alternatives, potentially using a longer path to reach a certain node, is stored and is accessible in a backtracking step. For simplicity, the handling of the stack in Algorithm 6.2 is omitted. Continuing the example from

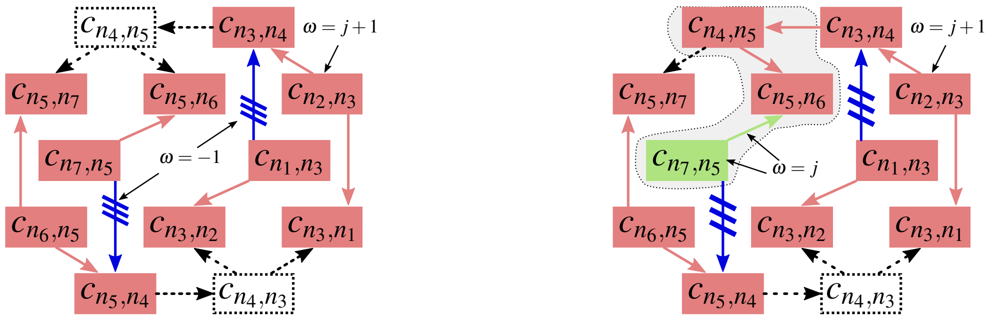
Figure 6.8: If Algorithm 6.2 reaches the impasse at node n_4, then it checks the stack of alternative routes to the nodes n_3 and n_5, and determines whether or not these can be used to reach n_4, see lines $26-43$ of Algorithm 6.5. For example, an alternative path to n_3, stored in the stack, is to use channel c_{n_2,n_3}. As Figure 6.8b illustrates in the upper right corner, the channels c_{n_2,n_3} and c_{n_3,n_4}, and the edge between them, already belong to the same acyclic subgraph identified by $\omega = j$. Therefore, using the channel dependency $(c_{n_2,n_3}, c_{n_3,n_4})$ is a valid alternative for the modified version of Dijkstra's algorithm to reach n_4. If multiple valid alternatives exist, then Nue routing selects the shortest among them, i.e., with respect to the weight/distance parameters of the channels. After a valid alternative is found for one island node, Algorithm 6.2 continues to operate as before to ensure that paths into clusters of island nodes, see the highlighted nodes below n_4 in Figure 6.8a, are calculated as well.

6.2.6.3 Using Formerly Isolated Nodes as Shortcut

The previous section explained how to use a local backtracking to solve routing impasses and how to find paths to an island node. Furthermore, these island nodes can be used to shorten the distance to previously discovered nodes. For instance, Algorithm 6.2 reaches an impasse for the network presented in Figure 6.8a and the local backtracking algorithm can find a path to node n_4, as described in Section 6.2.6.2. However, the nodes n_3 and n_5 have already been discovered and have a certain distance from the source node of the current routing step performed with Algorithm 6.2. Assume, the distance of node n_3 from the current source node is six hops and the distance of n_5 is nine hops when Algorithm 6.2 reaches the impasse. The local backtracking Algorithm 6.5 enables a valid path to n_4 via network node n_3. This former island node n_4 can now be used as a potential "shortcut" to reach node n_5, which shortens the distance of n_5 to eight hops.

However, existing channel dependencies have to be considered to make use of shortcuts during the routing within the complete channel dependency graph. While, in theory, it would be possible to invalidate decisions and dependencies for paths which use n_5 as an intermediate node, it would increase the runtime of the routing algorithm. Since the channel dependencies are built incrementally by Algorithm 6.2, changing an intermediate dependency (c_p, c_q) potentially invalidates all existing dependencies $(c_q, \cdot)$, as well as subsequent dependencies. To avoid the recalculation of paths, an approach is to incorporate the following optimization into Nue routing: Using islands as shortcuts is only allowed if existing local channel dependencies can be kept in place, which prevents the routing from reconsidering subsequent dependencies. Algorithm 6.6 shows how to verify whether a shortcut is possible or not.

Again, Figure 6.8a is used to exemplify the workings of Algorithm 6.6 to shorten previously calculated paths through former island nodes. Assume, after the impasse of Algorithm 6.2 and the solution for a path to the island node $n_4 = n_{c_q}$, the current state of $\overline{D}$ is shown in Figure 6.9. So, the first step is to check whether n_4 can be used as a shortcut to reach n_5 or not, i.e., $n_u = n_5$ in line 48. Hence, the algorithm must verify that changing the channel dependency $(c_{n_3,n_4}, c_{n_4,n_5})$ into the *used* state does not induce a cycle in $\overline{D}$. Assume further, that n_7 and n_5 were used as intermediate nodes to reach network node n_6 and



(a) State of $\overline{D}$ before shortcut **(b)** State of $\overline{D}$ after shortcut c_{n_4,n_5} found

Figure 6.9: The states of the channel dependency graph $\overline{D}$, of the subnetwork I^* of Figure 6.8a are shown; Figure 6.9a shows the state after solving the island problem for n_4; Figure 6.9b highlights the change to $\overline{D}$ when n_4 is used as a shortcut to reach n_5

Algorithm 6.6: Finding shortcuts through island nodes

```
Input: $I = G(N,C), \overline{D} = G(C,\overline{E}), (\cdot, n_{c_q}) =: c_q \in C$
Result: A shortcut through $n_{c_q}$ to adjacent nodes
// This is the expansion of Algorithm 6.5
if $n_{c_q}$ was an island then
    foreach $(n_{c_q}, n_u) \in C$ with $\emptyset \neq n_u.usedChannel \wedge c_q.distance + (n_{c_q}, n_u).weight < n_u.distance$ do
        $c_u \leftarrow (n_{c_q}, n_u)$                                        // possible shortcut
        $(c_q, c_u)$.state ← used
        if $\overline{D}$ is not cycle-free then
            Reset $(c_q, c_u)$.state
            continue
        List DependentChannels ← $\emptyset$
        foreach $n_v \in N$ with $(n_u, n_v) \in C$ do
            if $n_v.usedChannel = (n_u, n_v)$ then
                DependentChannels.add($n_v$.usedChannel)
        isShortcut ← true
        foreach $c_v \in DependentChannels$ do
            $(c_u, c_v)$.state ← used
            if $\overline{D}$ is not cycle-free then
                isShortcut ← false
                break
        if isShortcut then
            Reset $\omega$/state for previous $n_u$.usedChannel
            $n_u$.usedChannel ← $c_u$
        else
            foreach $c_v \in DependentChannels$ do
                Reset $(c_u, c_v)$.state
```

subsequent nodes, not shown in Figure 6.8a. Therefore, the usedChannel variable of n_6 is set to (n_5, n_6) and the channel dependency $(c_{n_7,n_5}, c_{n_5,n_6})$ is in the *used* state. The shortcut Algorithm 6.6 determines all dependent channels of n_5, i.e., all channels dependencies $(c_{\cdot,n_5}, c_{n_5,\cdot})$ calculated in the current routing step, see lines 54 – 57. Afterwards, the algorithm checks for all dependent channels $(n_5, \cdot)$ whether changing the dependency $(c_{n_4,n_5}, c_{n_5,\cdot})$ via the previous island induces a cycle or not. If no cycle is induced in $\overline{D}$, then node n_4 is a valid shortcut to reach node n_5 and shorten any subsequent path decisions. The usedChannel variable for n_5 can be changed to c_{n_4,n_5}, and previous changes to ω of c_{n_7,n_5} and $(c_{n_7,n_5}, c_{n_5,n_6})$ can be reversed.

6.2.7 Correctness, Completeness & Complexity

In the following, Nue routing is analyzed with respect to its destination-based and cycle-free characteristics, see Definition 3.4. The deadlock-freedom of Nue routing will be proven, as well. Meaning, the channel dependency graph D induced by the calculated paths is acyclic, and Nue calculates paths for each pair of nodes independently of the underlying network or the predefined number of available virtual channels. Hence, Nue can be applied to any arbitrary interconnection network, and many of the HPC network technologies listed in Section 1.1, i.e., even for the trivial 5-node ring visualized in Figure 6.1, which was not deadlock-free routable by any shortest-path routing algorithms. Subsequently, Proposition 6.1 summarizes Nue's time and memory complexity.

Lemma 6.2. *Nue routing is destination-based and cycle-free.*

Proof. Assume, Nue is not destination-based, and therefore the next channel c_{q+1} at a certain node is not unique for one destination and the following holds:

$$\exists n_x, n_y \in N \; \exists c_p, c_q \in C, c_p \neq c_q : R\big((\cdot, n_x), n_y\big) = \begin{cases} c_p \\ c_q \end{cases}$$

Algorithm 6.2 calculates the paths P_{n_y,n_x} for a source node n_x and all other nodes in the network in the opposite direction of the spanning tree created by the modified Dijkstra algorithm, i.e., that the paths follow the usedChannel variable towards the source node. The fact that the algorithm either assigns $\emptyset$ to usedChannel, see line 3, or one specific channel for each node, see line 21, contradicts the initial assumption. Hence, Nue routing is destination-based.

The fact that Nue is cycle-free follows directly from the fact that Nue is destination-based and the use of positive channel weights: Let P_{n_u,n_v} be a cyclic path, then either the node n_v is part of the cycle or not. The former case implies that n_v has a usedChannel $\neq \emptyset$ assigned to it by Algorithm 6.2, i.e., $\exists c_p, c_q \in C : c_p.\text{distance} + c_q.\text{weight} < 0 = n_v.\text{distance}$. This is only possible when weights are negative, which contradicts the fact that initial weights are positive and weight updates of channels are positive as well. In the later case, that this node n_v is not part of the cycle, then at least one channel $(\cdot, n_w)$ of the path has to contradict the destination-based property of Nue routing. Hence, Nue routing is cycle-free. □

The deadlock-freedom of Nue routing follows directly from its constant verification that using a channel for a path towards a destination does not induce a cycle in the channel dependency graph, as shown by Lemma 6.3.

Lemma 6.3. *Nue routing is deadlock-free.*

Proof. According to Theorem 3.1, the routing function is deadlock-free if and only if the corresponding channel dependency graph is acyclic. For each virtual layer, Nue creates a new complete CDG $\overline{D}_i$ and changes the states of the channels and channel dependencies of the escape paths to the *used* state. Since the escape paths are derived from a spanning tree, no cycle is induced in $\overline{D}_i$ after adding the escape paths. The performed cycle checks, see line 15 of Algorithm 6.2 (Section 6.2.4), line 35 of Algorithm 6.5 (Section 6.2.6.2), and lines 51 and 61 of Algorithm 6.6 (Sections 6.2.6.3), before any of the usedChannel variables are changed, are preventing Nue routing from creating a cycle in the acyclic $\overline{D}_i$. As a result, the complete CDG $\overline{D}_i$ for virtual layer L_i, for all $1 \leq i \leq k$, is cycle-free (Definition 6.1), and therefore Theorem 3.1 is applicable ensuring the deadlock-freedom. □

However, only ensuring deadlock-freedom for the calculated paths does not imply that Nue is actually able to calculate all required routes in the network, meaning the entire network state for the interconnect. Hence, Lemma 6.4 is required to show Nue's ability to ensures connectivity between any pair of network nodes, regardless of the given network topology or the provided number of virtual channels.

Lemma 6.4. *The Nue routing function ensures connectivity between any pair of two nodes in the interconnection network $I = G(N,C)$, i.e., $P_{n_u,n_v} \neq \emptyset \; \forall n_u, n_v \in N$, after the execution of Algorithm 6.3.*

Proof. Assume, there exists a pair of nodes, $n_u, n_v \in N$, for which the path $P_{n_u,n_v} = \emptyset$, i.e., Nue is incapable of calculating the path under the given virtual channel constraint. Therefore, either the network I is disconnected, which contradicts Definition 3.1, or the variable n_u.usedChannel $= \emptyset$ and the modified version of Dijkstra's algorithm, see line 24 of Algorithm 6.5, reaches an impasse. Due to the fact that $P_{n_u,n_v} = \emptyset$, the local backtracking algorithm falls back to the escape paths, see line 45 of Algorithm 6.5 (Section 6.2.6.2). Meaning, if $n_v \in N_i^d$, then from Definition 6.2 it follows that for every $n_w \in N$ there exists a $(n_w, \cdot) \in C_i^\tau$ with $R^\tau\big((n_w,\cdot), n_v\big) \in C^\tau$, i.e., a path $P_{n_u,n_v} = \big\{(n_u,\cdot), \ldots, (\cdot, n_v)\big\}$ exists using only channels in C_i^τ. This contradicts the initial assumption that $P_{n_u,n_v} = \emptyset$ holds, hence Nue routing ensures full connectivity. □

For the following complexity analysis, let Δ denote the maximum degree of the interconnection network $I = G(N,C)$, then it follows that $|C| \leq \Delta \cdot |N|$ and $|\overline{E}| \leq \Delta \cdot |C| \leq \Delta^2 \cdot |N|$. Most terms of the below shown Proposition 6.1 follow directly from the explanations provided in previous Sections 6.2.1 – 6.2.6.

After the partitioning of the destination nodes, which is calculated in time complexity $\mathcal{O}\big(|C|\big)$ with the multilevel k-way partitioning algorithm, see Section 6.2.5, Nue routing initializes the current complete channel dependency graph $\overline{D}_i$ for the path calculation within virtual layer L_i. The generation of $\overline{D}_i$ can be performed in $\mathscr{O}\big(\Delta \cdot |C_i|\big)$ and the complete CDG requires $\mathscr{O}\big(|N| + |C_i| + |\overline{E}_i|\big)$ memory, because it includes some information of the nodes of I as well.

The convex subgraph for the subset $N_i^d \subseteq N$ of destinations, see Section 6.2.3, can be calculated with a time complexity $\mathscr{O}\big(|N_i^d| \cdot (|N| + |C|)\big)$, because each network node of I is visited only once during the forward step and once during the backward step. Assuming the worst case, i.e., $N_i^d \approx N$, yields a time complexity of $\mathscr{O}\big(\Delta \cdot |N|^2)\big)$ for each virtual layer. The memory complexity to calculate the convex subgraph, with Algorithm 6.1, is $\mathscr{O}\big(|N| + |C|\big)$.

As mentioned in Section 6.2.3, Brandes' algorithm computes the betweenness centrality values for each node of H_i in $\mathscr{O}\big(|N_i^H| \cdot |C_i^H|\big)$ and requires $\mathscr{O}\big(|N_i^H| + |C_i^H|\big)$ memory [Bra01, Theorem 8]. The convex subgraph H_i for N_i^d is bound by I, i.e., the network itself. Hence, the time complexity for Nue routing to identify the most central node $n_{r,i} \in N$, with respect to N_i^d, can be approximated with $\mathscr{O}\big(|N| \cdot |C|\big)$ and the memory complexity can be approximated with $\mathscr{O}\big(|N| + |C|\big)$.

The computation of the spanning tree rooted at $n_{r,i}$ for the escape paths can be accomplished with the original version of Dijkstra's algorithm for the network I. Meaning, the time complexity of this step is $\mathscr{O}\big(|N| \cdot \log |N| + |C|\big)$ when a heap with $\mathscr{O}(1)$ time complexity for the "decrease-key" operation is used, e.g., a Fibonacci heap. The corresponding memory complexity is $\mathscr{O}\big(|N| + |C|\big)$. This completes the initialization phase for one virtual layer L_i.

The time complexity, excluding the acyclicity check, for the modified version of Dijkstra's algorithm, as presented in Algorithm 6.2, is $\mathscr{O}\big(|C_i| \cdot \log |C_i| + |\overline{E}_i|\big)$ when executed on a complete CDG $\overline{D}_i$. As previously mentioned, a heap with $\mathscr{O}(1)$ time complexity for the "decrease-key" operation, is needed to achieve this complexity. The optimization, see Section 6.2.6.1 and conditions (i) – (iv), results in a differentiated time complexity for the acyclicity check of $\overline{D_i}$, see line 15 of Algorithm 6.2. In the best case, i.e., condition (i) or (ii) are applied, the time complexity is $\mathscr{O}(1)$. The same applies to the "merge" of the two acyclic subgraphs assuming that $\omega(c_q)$ is zero, see line 6 of Algorithm 6.4. An actual merge, see lines 7 and 8, of two vertex-disjoint and acyclic subgraphs with varying identification numbers can only be performed $|N|$ times throughout the execution of Nue routing. Hence, the time complexity of all merge steps is at most $\mathscr{O}\big(|N| \cdot (|C_i| + |\overline{E}_i|)\big)$. The time complexity for any depth-first search within a subgraph of $\overline{D_i}$ is $\mathscr{O}\big(|C_i| + |\overline{E}_i|\big)$. However, the DFS has to be executed at most once per edge $e \in \overline{E}_i$ and per virtual layer L_i, because afterwards the state of e is either *used* or *blocked*, and condition (iv) cannot be applied again. Therefore, the aggregated time complexity for $|N|$ executions of Algorithm 6.2, i.e., once for each destination node, is $\mathscr{O}\big(|N|(|C_i| \log |C_i| + |\overline{E}_i|) + (|N| + k|\overline{E}_i|) \cdot (|C_i| + |\overline{E}_i|)\big)$. The memory complexity for the modified version of Dijkstra's algorithm is $\mathscr{O}\big(|N| + |C_i| + |\overline{E}_i|\big)$. The DFS to search for cycles can be performed within $\overline{D}_i$, and therefore does not increase the memory complexity.

Considering the entire routing function, then the time complexity, as well as the memory complexity, to store the paths in forwarding tables is $\mathscr{O}\big(|N|^2\big)$, and the time complexity to update the channel weights is $\mathscr{O}\big(|N|^2\big)$. The summary of the complexity analysis of the individual steps for Nue routing is given in the following proposition:

Proposition 6.1. *The time complexity of Nue routing for a given interconnection network I, with a fixed maximum switch radix* Δ*, with* $\Delta \in \mathbb{N}$*, and a fixed number of supported virtual channels k, with* $k \in \mathbb{N}$*, is*

$$\begin{aligned}
&\mathscr{O}\big(|C|\big)_{\textit{Partitioning}} \\
&+ k \cdot \mathscr{O}\big(\Delta|C_i|\big)_{\textit{Build complete CDG}} \\
&+ k \cdot \mathscr{O}\big(\Delta|N|^2\big)_{\textit{Convex } H_i} \\
&+ k \cdot \mathscr{O}\big(|N|\log|N| + |C|\big)_{\textit{Spanning Tree}} \\
&+ \mathscr{O}\big(|N|(|C_i|\log|C_i| + |C_i| + |\overline{E}_i|) + k|\overline{E}_i|(|C_i| + |\overline{E}_i|)\big)_{\textit{Routing}} \\
&+ \mathscr{O}\big(|N|^2\big)_{\textit{Forwarding Tables}} \\
&+ \mathscr{O}(|N|^2)_{\textit{Weight Updates}} \\
&= \mathscr{O}\big(|N|^2(\Delta\log(\Delta|N|) + k\Delta^4)\big) \\
&= \mathscr{O}\big(|N|^2 \cdot \log|N|\big)
\end{aligned}$$

while its memory complexity (including storing the result) is

$$\mathscr{O}\big(|N| + \Delta \cdot |N| + \Delta^2 \cdot |N| + |N|^2\big) = \mathscr{O}\big(|N|^2\big)$$

assuming that the number of channels of I can be approximated by $\Delta \cdot |N|$ *and assuming that* $|\overline{E}_i| \leq \Delta^2 \cdot |N|$.

Therefore, Nue routing achieves one of the initial goals stated in Table 2.1, i.e., to have a time complexity less or equal to $\mathscr{O}\big(|N|^3\big)$. Section 6.3.3 will present Nue's runtime in comparison to other algorithms, showing the competitiveness of the developed Nue implementation.

6.3 Evaluation of Nue Routing

For the following sections, an implementation of Nue routing has been added to the InfiniBand subnet manager, which allows simulations and measurements for the InfiniBand network technology. These simulations rely on the toolchain presented in Chapter 4 enabling the collection of edge forwarding index statistics for various topologies and routings, measurement of their runtimes, as well as flit-level accurate throughput measurements by use of the OMNeT++ framework. Section 6.3.1 compares Nue routing to other routing algorithms with respect to the path length, since Nue does not necessarily create shortest paths in the network, and with respect to the induced edge forwarding index. Subsequently, throughput simulations for the all-to-all exchange pattern, recall Section 4.3.3.2, are carried out in Section 6.3.2 for different topologies routed with routing algorithms implemented in the production-quality InfiniBand network manager, OpenSM in version 3.3.19 [Mel13]. Most of these routing algorithms either omit the calculation of switch-to-switch paths entirely, e.g., fat-tree routing (Section 3.3.5) or Up*/Down* routing (Section 3.3.4), or ignore these paths during the deadlock-avoidance calculation phase, e.g., DFSSSP (Section 3.3.8), since switch-to-switch paths are only used for management messages and not data messages. Therefore, for a fair comparison, switches are excluded from the set of destination nodes for Nue routing while performing the following evaluations. Furthermore, the Nue implementation in OpenSM forgoes the usage of a Fibonacci heap, and uses d-ary heaps, with $d = 4$, instead, which outperform a Fibonacci heap in practice despite their $\mathcal{O}(\log_d n)$ time complexity for the “decrease-key” operation [Joh75; LST14]. Finally, Section 6.3.3 shows the fault-resiliency and runtime of Nue routing, indicating its supremacy over other state-of-the-art topology-agnostic routing algorithms.

6.3.1 Path Length and Edge Forwarding Index

Nue routing does not always create routes along the shortest-paths, depending on the available number of virtual channels and whether the escape path is used or not. However, non-minimal routes increase latency and may cause higher network congestion. So, analyzing the path length of routes created by Nue in comparison to other shortest-path algorithms is reasonable. Additionally, the edge forwarding index γ for inter-switch ports in the network, see Definition 3.12, is investigated for these topology and routing combinations. The latter metric demonstrates the quality of the routing function in terms of path balancing. Remember that a high minimum $\gamma_R(c)$ and low maximum $\gamma_R(c)$, for all $c \in C$, are indicators for a well balanced routing algorithm.

To evaluate both metrics, the simulation toolchain is used to create 1,000 random topologies. Each topology consists of 125 switches interconnected by 1,000 channels, and eight terminals connected to each switch. Nue routing is applied, with $1 \leq k \leq 8$ virtual channels, as well as LASH and DFSSSP routing, to create deadlock-free forwarding tables. The toolchain reports the following metrics for each topology/routing combination: minimum, maximum, and average edge forwarding index, and the standard deviation (sdv), i.e., $\gamma_{\min}(I_t, R) := \min\limits_{c \in C} \gamma_R(c)$, $\gamma_{\max}(I_t, R)$, $\gamma_{\text{avg}}(I_t, R)$, and $\gamma_{\text{sdv}}(I_t, R)$ for topology I_t and routing R. These metrics are then arithmetically averaged for all 1,000 topologies:

$$\Gamma_{\min}^{R} = \frac{\sum\limits_{t=1}^{1{,}000} \gamma_{\min}(I_t, R)}{1{,}000}$$

and so forth.

Figure 6.10 shows the result as box plots, with whiskers indicating $\Gamma_{\min}^{R}$ and $\Gamma_{\max}^{R}$, and with the box indicating the average Γ_{avg}^{R} and $\Gamma_{\text{avg}}^{R} \pm \Gamma_{\text{sdv}}^{R}$. As can be seen, Nue routing performs almost similar to DFSSSP, for $k \geq 4$. It is worth mentioning, that DFSSSP needs at least four virtual channels to calculate deadlock-free routes for all these topologies, or even five virtual channels in some exceptional cases. LASH’s virtual channel requirement is lower compared to DFSSSP and ranges between two and four VCs. However both, Nue and DFSSSP, clearly outperform LASH with respect to the edge forwarding index metric. Even so Figure 6.10 indicates that DFSSSP slightly outperforms Nue, it is evident that Nue routing is designed for arbitrary topologies and supports every given number of virtual channels. Therefore, Nue will be able to calculate deadlock-free paths even if when scaling up the size of the topologies, while the other routings, such as DFSSSP, will fail due to VC limitations, see Section 6.3.3.

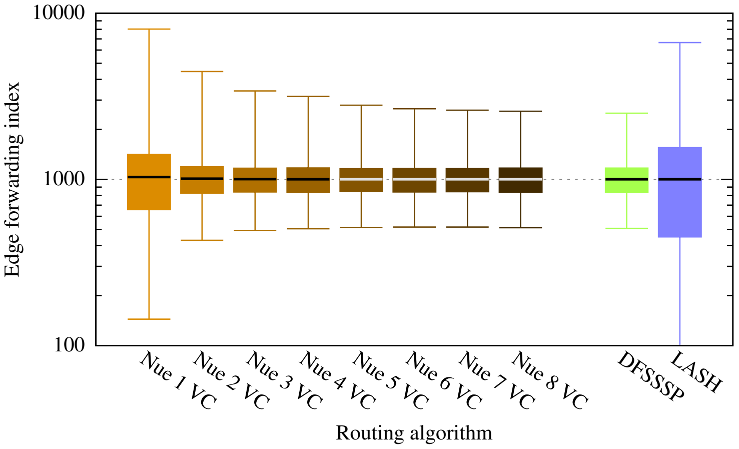


Figure 6.10: Averaged edge forwarding index metrics shown in whisker plot: minimum Γ^{R}_{min} and maximum Γ^{R}_{max} for the extremes, and average Γ^{R}_{avg} (± standard deviation Γ^{R}_{sdv}) for the box; Simulations for 1,000 random topologies each consisting of 125 switches, 1,000 terminals, and 1,000 switch-to-switch channels

The increased edge forwarding indices for $k < 4$ have two reasons: firstly, the concentration of paths on certain channels to bypass routing restrictions, and the other reason are longer paths due to the use of the escape paths or parts of them. It is clear that a longer path changes $\gamma_R(c)$ for more ports or channels in the network. Hence, for all 1,000 random topologies, the maximum path length in the network was determined, as well. For the best case, Nue routing needs only two virtual channels to support the same maximum path length as the shortest-path algorithms DFSSSP and LASH. On average, Nue routing achieves the same maximum path length as DFSSSP, i.e., an arithmetic average of 5.3, if Nue distributes the paths among at least seven virtual layers. The worst case length of the longest path for Nue ranges between 7 hops and 10 hops, depending on the given number of VCs, while it is 6 for DFSSSP and LASH routing.

The number of fall backs to escape paths depends on many factors, such as topology type, size, number of virtual channels, and the chosen root node for the spanning tree. For the random topologies with no additional virtual channels, i.e., the first whisker marked Nue 1 VC, Nue did fall back for 0% – 9.7% of the destinations, with an average of 0.95% across all 1,000 simulations for this case. For 8 VCs this average is below 0.006%. A general prediction of the number of times Nue uses the escape paths is beyond the scope of this thesis.

6.3.2 Throughput for Regular and Irregular Topologies

Additionally to the random topologies from Section 6.3.1, the achievable throughput for four standard topologies, i.e., fat-tree, torus, Kautz graph, dragonfly (see Section 3.2 for details about all these topologies), is measured with the simulation toolchain. Three real-world topologies, namely Cray's Cascade (Section 3.2.4), the multi-island fat-tree of Taurus, and TSUBAME2's 2nd rail fat-tree network connecting 1,408 compute node (Section 3.2.6), are analyzed. The Cascade topology is configured

Table 6.1: Topology configurations, with link redundancy $r_{C'}$, used for throughput simulation results visualized in Figure 6.11

Topology	Switches	Terminals	Inter-switch channels	$r_{C'}$
Random	125	1,000	1,000	1
3D torus(6,5,5)	150	1,050	1,800	4
10-ary 3-tree	300	1,100	2,000	1
Kautz(7,3)	150	1,050	1,500	2
Dragonfly(12,6,6,15)	180	1,080	1,515	1
Cascade(2 groups)	192	1,536	3,072	1
TSUBAME2	243	1,408	3,384	1
Taurus	211	2,090	2,490	1

with 192 global channels to connect two Cascade electrical groups. Furthermore, an arbitrary random topology has been chosen among the 1,000 created random topologies for this throughput measurement.

All topologies accommodate approximately 1,000 terminals to allow comparison between topologies, as well. Each switch is connected to at least one terminal for all topologies, except for the two fat-tree topologies. Detailed topology configurations, utilizing 36-port switches (exception: 48 ports for Cascade), are given in Table 6.1. The simulated interconnects are assumed to be based on QDR InfiniBand with a limit of eight virtual channels. Hence, the original QDR/FDR multi-island fabric of Taurus is downgraded to QDR for this simulation, and each HCA is assumed to be a terminal, raising the number of terminal above the number of nodes due to the dual-HCA I/O and GPU nodes. The remaining simulation parameters are in compliance with the configuration used in Section 4.3.2. The redundancy $r_{C'}$ listed in Table 6.1 refers to a multiplication of switch-to-switch channels with respect to the usual topology definition to increase the port usage, see Definition 3.3.

The flit-level simulator performs an all-to-all send operation with 2 KiByte message size between all terminals of the network. An exchange pattern of varying shift distances, see Section 4.3.3.2, is used at each terminal to communicate with all other terminals. Simulating uniform random injection traffic yields similar behaviour of Nue, and showing these results will not provide further insight. The throughput is measured for all nine routing algorithms that are available in OpenSM version 3.3.19, see Section 3.3 for the complete list. Impossible topology and routing combinations, such as Torus-2QoS routing for the 10-ary 3-tree, are ignored. Nue routing is compared for a given topology to the other usable algorithms, for every number of virtual channels $1 \leq k \leq 8$, see Figure 6.11.

Besides the simulated throughput for each topology and routing combination, the number of needed virtual channels is given atop of the bars in Figure 6.11. For example, DFSSSP routing needs four VCs for a deadlock-free routing of the random topology. However, the deadlock-free SSSP routing usually uses all eight available virtual channels to optimize the path balancing across the virtual layers, a technique to increase the throughput slightly [Dal90].

Two trends are visible for all investigated topologies: First, an increase of used virtual channels

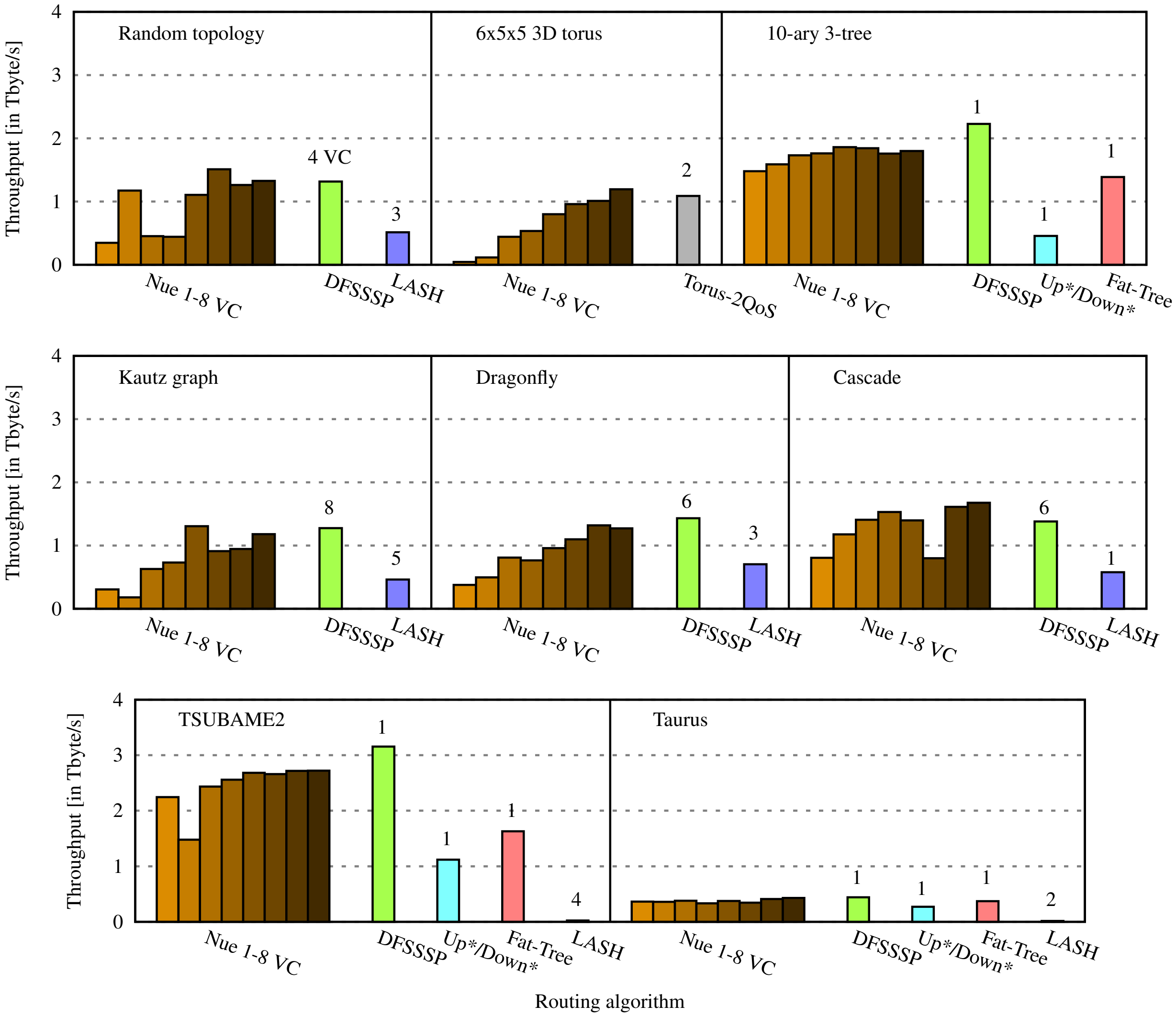


Figure 6.11: Simulated throughput for an all-to-all operation on five standard and three real-world topologies, as configured according to Table 6.1; Nue routing shown for all numbers of VCs between 1 and 8 (from left to right); VC requirement by other routings for deadlock-freedom are shown atop the individual bars

for Nue also increases the throughput for the all-to-all communication. This is a result of the decreased $\gamma_{\max}(I_t, \text{Nue})$ when multiple virtual channels are used, as reported in Section 6.3.1. The outliers from this pattern, e.g., the decrease in throughput for the random topology and four virtual channels, correlate to a sudden increase in fall backs to the escape paths. In this particular example, Nue had to fall back for 14 of the 1,000 destinations, while Nue with two and more than four virtual channels was able to route the topology without any fall back.

The second trend is that Nue shows a slight variance in throughput after reaching a certain peak, usually for about $k \geq 5$ in these examples, but this generally depends on topology type and size. This behavior can be accounted to a mismatch between the static routing and the execution order of the point-to-point communications, which assemble the all-to-all traffic pattern. A mismatch can cause temporary congestion in the network which slows down the entire communication process as a result, which is a known problem, see Section 4.4.3 or Hoefler et al. [HSL08], respectively.

In general, Figure 6.11 shows that Nue routing is competitive to the best performing routing for each individual topology, i.e., offers between 83.5% for the 10-ary 3-tree and 121.4% throughput for the Cascade network in comparison. Occasionally, depending on the given number of virtual channels, Nue is able to outperform the best competitor, which are Torus-2QoS routing for the torus, and DFSSSP routing for all other topologies. For example, for the random topology, Nue with $k \geq 6$ offers up to 15% higher throughput than DFSSSP, and for the Cascade network up to 21% higher throughput, with $k \geq 3$ (excluding $k = 6$). Furthermore, when given enough virtual channels, Nue is able to outperform the other routings, such as fat-tree routing or LASH, to a great extent. Therefore, it is evident that Nue routing is an adequate alternative to the other investigated algorithms, or is at least a suitable fall back in case the best performing algorithm becomes inapplicable, as shown in the following Section 6.3.3.

6.3.3 Runtime and Practical Considerations

Proposition 6.1 mathematically derived an $\mathcal{O}\left(|N|^2 \cdot \log|N|\right)$ time complexity of Nue routing. To put this into perspective, the runtime of Nue, with eight virtual channels, needs to be compared to other deadlock-free routing algorithms implemented in OpenSM. Slight variations in the runtime, depending on the given number of virtual channels, are possible for Nue, as for the other routing algorithms. However, providing these numbers here will not provide substantially more insight. For the following measurement, an implementation of Nue routing algorithm (Algorithm 6.3) has been integrated into the current OpenSM, i.e., version 3.3.19, to achieve a fair comparison.

The underlying base topology for this runtime measurement is a 3-dimensional torus, which has proven to be challenging for most investigated fault-tolerant algorithms, see Section 4.4.3, and therefore offers insight into the fault-tolerance and applicability of Nue, at the same time. The simulation toolchain is used to create 25 3D torus networks with a difference in dimension of at most one, i.e., the grid size for the switches starts at 2x2x2, 2x2x3, 2x3x3, …, and goes up to 10x10x10. Each of these switches connects to four terminals, hence the 10x10x10 torus accommodates 4,000 terminals. Furthermore, the

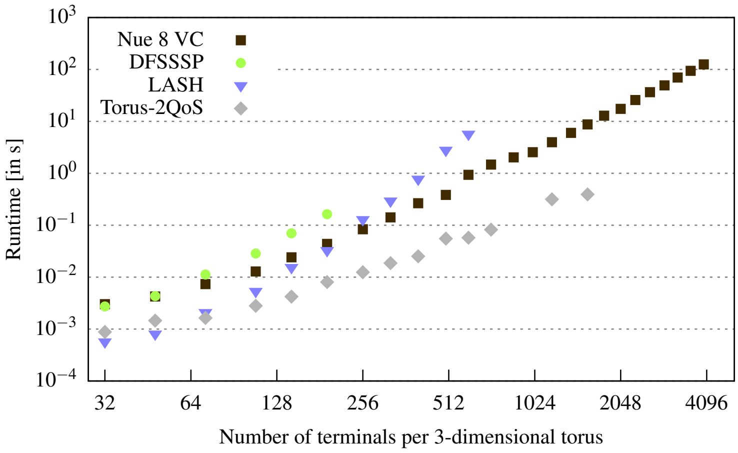


Figure 6.12: Runtime comparison of deadlock-free routings for 3-dimensional tori (without channel redundancy) of various topology sizes with 1% injected link failures; Four terminals per switch (e.g., smallest: 2x2x2 torus with 32 terminals, largest topology: 10x10x10 torus with 4,000 terminals); Missing dots: routing failed

assumption for this test is that the maximum of available virtual channels is 8, adjacent switches utilize no channel redundancy, and 1% link/channel failures are randomly injected into the topology. The 1% link failures have been chosen according to the observed annual failure rate of production HPC systems, see Section 4.1. Besides Nue, the toolchain is used to evaluate the runtime of DFSSSP, LASH, and Torus-2QoS routing to calculate deadlock-free routing tables for the same faulty topology. The testbed, which executes the subnet manager, is a dual-socket Intel Xeon E5-2620 server with 64 GiByte of main memory. The InfiniBand fabric simulator (ibsim) is pinned to socket 0 and the OpenSM is pinned to socket 1 to minimize disturbances.

Two main conclusions can be drawn from the results shown in Figure 6.12: First, Nue is competitive in terms of runtime. Nue routing calculates the forwarding tables faster than the topology-agnostic DFSSSP, which has the same time complexity of $\mathcal{O}\left(|N|^2 \cdot \log|N|\right)$, see Table 2.1 or previously published work by the author of this thesis [DHN11], respectively. Nue outperforms the runtime of LASH routing for tori larger than 4x4x4 with 256 terminals attached. Only the topology-aware Torus-2QoS routing is on average 9x faster than Nue, which is as expected since Torus-2QoS is able to avoid deadlocks analytically. The second important result is an applicability of 100%, i.e., Nue routing scales with the topology size. The other three deadlock-free algorithms fail, notice the missing data points in Figure 6.12, either because the algorithms run out of virtual layers, i.e., DFSSSP and LASH exceed the 8 VC limit, or because the failures prevent an analytical solution for the deadlock-free paths problem, in the case of Torus-2QoS routing. Only Nue routing is always applicable while offering good path balancing.

7 Summary, Conclusion, and Future Work

The most important role of each supercomputer is to provide other research domains, such as material science, meteorology and climate research, biomedical informatics, or astrophysics, etc., with computational resources otherwise not available. The high-speed interconnection network, connecting the nodes of the HPC system, plays a crucial role in achieving scalability and throughput of these scientific applications. Key factors of the interconnection network are low latency, high bandwidth, high message throughput, deadlock-freedom, resiliency, and availability, hence factors which depend not only on the networking hardware, but also depend on the software used for network management. This thesis advances the state-of-the-art of network management in three main areas of interest for the HPC community, by:

- Showing that HPC interconnection networks can be operated in fail-in-place mode and providing a simulation toolchain to plan future systems and to establish operation policies,
- Developing a new and low overhead scheduling-aware routing algorithm to improve the scientific throughput and network utilization of multi-user/multi-job HPC environments, and
- Introducing a novel concept for solving the deadlock-freedom of routing algorithms, which allows the calculation of deadlock-free and path-balanced forwarding rules for arbitrary topologies within a given virtual channel constraint.

The developed simulation toolchain helps system designers and administrators in the decision making about the correct combination of topology and routing algorithm, and helps to extrapolate the performance degradation of a fail-in-place network based on estimated or known failure rates. The comprehensive empirical study of combinations of topologies, routing algorithms, and network failures shows that HPC networks, and not only hard drives, can be operated in fail-in-place mode to reduce maintenance costs and to utilize the remaining resources. Even so the resulting irregular topologies pose a challenge to the used routing algorithms, a low failure rate of the network components supports a fail-in-place network design strategy, which bypasses non-critical failures during the system's lifetime. To guarantee high overall network throughput over the lifetime of the supercomputer, a resilient topology needs to be combined with an appropriate fault-tolerant or topology-agnostic routing algorithm. However, both types of existing deterministic routing algorithms, i.e., fault-tolerant topology-aware and topology-agnostic, show limitations. The performance degradation using a topology-aware routing algorithm increases more with an increase of failures compared to topology-agnostic routings. Furthermore, a large number of switch and link failures can deform a regular topology to a state which cannot be compensated by the topology-aware routing, because it cannot recognize the underlying topology. In contrast, topology-agnostic routing algorithms are ideal for fail-in-place networks by design, but the investigated algorithms either show weaknesses in terms of deadlocks, such as MinHop routing, or cannot be used beyond a certain network size. The deadlock-avoidance via virtual channels by state-of-the-art routings can require more than the available number of virtual channels, e.g., in the case of LASH and DFSSSP for the 3-dimensional torus(7,7,7) with more than 2,000 terminals, both exceed the current hardware limit of eight VCs. The performed analysis of lost connections after a failure, and time before a rerouting step is finished, is valuable for other fault tolerance areas as well. The knowledge about the runtime of a rerouting step can be used to calculate appropriate retry counters and timeout values in the communication layer.

While the fail-in-place network operation increases the resiliency and availability considerably, the overall achievable throughput decreases slightly when the supercomputer is used by a single scientific application. Nonetheless, this is rarely the case, since most HPC systems located at universities or national laboratories are used by multiple users simultaneously. When the network of these systems utilizes flow-oblivious routing, e.g., as used by the 37.6% IB-based supercomputers of the TOP500 list, then this causes two problems: parts of the network are underutilized, while application performance is reduced by local congestion. The developed scheduling-aware routing (SAR) approach for the interconnection network is able to mitigate these problems. The SAR approach periodically analyzes the state of the batch system, with its low overhead filtering tool, and reroutes the network while factoring in the locality of concurrently running applications. As a result, the observable throughput for MPI-parallelized applications improves considerably, and is less sensitive to actual locality of the compute nodes within the supercomputer. Furthermore, the amount of dark fiber, i.e., installed but unused cables, is reduced. Combining the knowledge of two different resource management domains has proven to be effective in theory, as well as in practice in over one year of usage on the Taurus HPC system. The conducted empirical study, based on the simulative replay of the historical job-to-node allocation of Taurus and TSUBAME2, reveals that SAR is able to improve the overall system utilization by up to 17.74%, and reduce the effective edge forwarding index of individual jobs by up to 71.2%. Conducted communication benchmarks on Taurus, while the system was simultaneously used by other researchers and scientific applications, showed throughput increases by up to 17.6% after SAR optimized the forwarding rules for the locality of the benchmark job. Unfortunately, changing the forwarding rules, not subject to error correction, within an interconnection network can cause unintended network packet delivery, such as out-of-order delivery, packet drops, and deadlocks, etc. Upper layer protocols, e.g., the MPI communication protocol, might fail, if a lossless interconnection technology, such as InfiniBand, is not able to handle these problems. The devised five-phase update protocol, tailored for the InfiniBand architecture, achieves a property preserving network update to ensure fault-free operation. Even so, the update protocol is currently inapplicable due missing firmware features, users of Taurus have not experienced any application crashes due to SAR, presumably because of resiliency features build into the transport layer of InfiniBand.

Additionally, a fail-in-place network is also more prone to deadlocks, when it is based on lossless interconnection technology. The lossless transmission of messages, e.g., by using credit based flow control in InfiniBand or Priority Flow Control (PFC) with pause frames in Ethernet, requires deadlock-free routing to function reliably. The same is true for many Network-on-Chip architectures. However, the applicability of topology-aware and topology-agnostic routings for faulty regular topologies or arbitrary topologies is limited, as discussed before, and universally applicable routings, such as Up*/Down*, are scarce and perform poorly with respect to path length and balancing. The novel approach of applying a graph search algorithm within the complete channel dependency graph instead of the actual network, presented in this thesis, is the first reliable strategy to route arbitrary topologies with a limited number of virtual channels, while achieving close to optimal path length and competitive path balancing. Similar to Up*/Down*, deadlock-free forwarding rules can always be calculated, even in the absence of virtual channels. A model implementation of the new approach, called Nue routing, is tailored for the deadlock-free, oblivious, and destination-based routing needed in Converged Enhanced Ethernet (CEE) and InfiniBand and can directly be employed for both, e.g., using InfiniBand's virtual lanes or using PFC together with Priority Code Point for CEE. Possible applications of Nue for NoC architectures include, but are not limited to, the routing between tiles connected by virtual channel routers in a fault-tolerant manner. An extensive study, conducted for multiple regular, irregular, and real world topologies, shows that Nue routing outperforms other topology-agnostic routings, such as LASH and DFSSSP, with respect to runtime and resiliency. Flit-level simulations indicate that Nue enables competitive or superior global throughput, i.e., between 83.5% and 121.4% throughput for an all-to-all traffic pattern, in comparison to the best performing state-of-the-art routing for the respective topology. Another advantage is Nue's capability of arbitrarily limiting the number of used VCs which allows the combination of deadlock-freedom and quality of service (QoS) when QoS is based on the same technology feature. For instance, InfiniBand's service levels are mapped to virtual lanes, which are all used by LASH or DFSSSP to avoid deadlocks. So, previously, the choice for IB was either having QoS with topology-aware routing or ignoring QoS and using topology-agnostic routings based on virtual lanes. All these characteristics and advantages, combined with Nue's low time complexity of $\mathcal{O}(|N|^2 \cdot \log |N|)$ and memory complexity of $\mathcal{O}(|N|^2)$, make Nue routing a suitable algorithm to route modern large-scale supercomputers, lossless data center networks, and NoC architectures.

The developed simulation toolchain, as well as implementations of the two novel routing approaches, i.e., scheduling-aware routing and Nue routing, are freely available:

- Topology generator and fail-in-place simulation toolchain:
  `http://spcl.inf.ethz.ch/Research/Scalable_Networking/FIP`
- Scheduling-aware routing and filtering interface between SLURM and OpenSM:
  `https://gitlab.com/domke/osm-routing-dev/tree/sar-3.3.20`
- Implementation of Nue routing in OpenSM for IB-based networks:
  `https://gitlab.com/domke/osm-routing-dev/tree/nue-3.3.19`

allowing researchers, systems designers, and administrators, to conduct further studies or to improve their supercomputers.

Both devised routings, SAR and Nue, are opening up new research areas for the HPC community to adapt and improve the underlying ideas. These algorithms did undergo extensive simulations with various topologies and real-world testing on two production supercomputers in preparation for an integration into the network manager of the OpenFabrics Enterprise Distribution, which would make the routings available on all IB-based HPC system. Especially, Nue should be evaluated against MinHop routing as the fall back routing algorithm, since MinHop fails to provide the required deadlock-freedom. Additional future work for network hardware vendors and the HPC community consists of revising the support to drain queue pairs and adapting the five-phase update protocol for other upper layer protocols, besides MPI.

While the most accurate results are provided by the flit-level simulation framework, the underlying combination of OMNeT++ and ibmodel currently lacks scalability for networks beyond 5,000 terminals due to the single-threaded and memory-inefficient implementation. One option for future work is to parallelize the ibmodel and reduce the memory footprint considerably. An alternative is the renunciation of flit-level accuracy and the replacement of the 4th stage of the toolchain, i.e., the simulator, with the highly parallel CODES simulator, enabling packet-level simulations instead. Furthermore, the number of supported base topologies by the toolchain can be expanded, to include novel designs, such as ExpressMesh, Slim Fly, flattened butterfly, and HyperX networks.

Evidently, the scheduling-aware routing algorithm decreases the theoretical upper bound for the worst-case congestion within batch jobs, and therefore improves the runtime of all-to-all communication patterns substantially. However, unknown is both the impact of SAR on the mutual interference of multiple jobs and the available throughput for deviating communication patterns. Hence, future research is required to shed light on these questions, and to potentially incorporate additional information, either provided by users or provided by the system, into the routing decisions to enhance the quality of SAR. Furthermore, a yet uninvestigated field is the applicability of SAR to optimize the default paths of an adaptively routed network architecture to minimize the number of adaptations performed by switches.

A wide range of directions for future research opens up with the novel approach of routing within the complete channel dependency graph, e.g., most obviously: combining SAR and Nue routing into an unified algorithm. In addition, some interconnection architectures require source-based routing or all-to-all routing (i.e., paths are determined by source and destination), as used in Myrinet or Bull's eXascale Interconnect (BXI), respectively, which potentially benefit from the new concept of avoiding routing deadlocks. Whether this concept is applicable to architectures which utilize distributed routing or not is as unknown as a parallelized algorithm to verify the cycle-freedom of the complete CDG to improve Nue's runtime. Furthermore, Nue's distribution of destinations to virtual layers requires additional research to decrease the fall backs to the escape paths, and therefore improving latency and achievable throughput. Logically, Nue's ability to route arbitrary topologies in the absence of virtual channels renders it a suitable routing to calculate the escape channels for fully adaptive routings based on Duato's protocol [DT03, 14.3.2], usable in Intel's Omni-Path or Cray's Cascade architectures, and the integration of Nue into the respective network manager should be evaluated. The same holds for NoCs and software-defined networking (SDN) architectures, where a comparison to the state-of-the-art routings is desired.

Bibliography

[AA13] A. B. Ahmed and A. B. Abdallah. „Architecture and Design of High-throughput, Low-latency, and Fault-tolerant Routing Algorithm for 3D-network-on-chip (3D-NoC)“. In: *J. Supercomput.* 66.3 (Dec. 2013), pp. 1507–1532. ISSN: 0920-8542. DOI: `10.1007/s11227-013-0940-9`.

[AAS87] G. Adams III, D. Agrawal, and H. Siegel. „A Survey and Comparision of Fault-Tolerant Multistage Interconnection Networks“. In: *Computer* 20.6 (June 1987), pp. 14–27. ISSN: 0018-9162. DOI: `10.1109/MC.1987.1663586`.

[Adi+05] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. „Blue Gene/L Torus Interconnection Network“. In: *IBM J. Res. Dev.* 49.2 (Mar. 2005), pp. 265–276. ISSN: 0018-8646. DOI: `10.1147/rd.492.0265`.

[Ahn+09] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. „HyperX: Topology, Routing, and Packaging of Efficient Large-scale Networks“. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. New York, NY, USA: ACM, 2009, 41:1–41:11. ISBN: 978-1-60558-744-8. DOI: `10.1145/1654059.1654101`.

[Aji+12a] Y. Ajima, T. Inoue, S. Hiramoto, T. Shimizu, and Y. Takagi. „The Tofu Interconnect“. In: *IEEE Micro* 32.1 (2012), pp. 21–31. ISSN: 0272-1732. DOI: `10.1109/MM.2011.98`.

[Aji+12b] Y. Ajima, T. Inoue, S. Hiramoto, and T. Shimizu. *Whitepaper: Tofu: Interconnect for the K computer*. Tech. rep. Vol. 48, No. 3. Fujitsu, July 2012, pp. 280–285.

[AK11] D. Abts and J. Kim. *High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011. URL: `http://dx.doi.org/10.2200/S00341ED1V01Y201103CAC014`.

[Alv+12] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. *Whitepaper: Cray® XC™ Series Network*. Tech. rep. WP-Aries01-1112. Cray Inc., Nov. 2012, p. 28. URL: `http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf` (visited on 03/16/2016).

[Ari+10] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. „The PERCS High-Performance Interconnect“. In: *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*. Aug. 2010, pp. 75–82. DOI: `10.1109/HOTI.2010.16`.

[ARK10] R. Alverson, D. Roweth, and L. Kaplan. „The Gemini System Interconnect“. In: *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*. Aug. 2010, pp. 83–87. DOI: `10.1109/HOTI.2010.23`.

[ATB14] A. H. Abdel-Gawad, M. Thottethodi, and A. Bhatele. „RAHTM: Routing Algorithm Aware Hierarchical Task Mapping“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 325–335. ISBN: 978-1-4799-5500-8. DOI: `10.1109/SC.2014.32`.

[Bad99] D. A. Bader. *A Practical Parallel Algorithm for Cycle Detection in Partitioned Digraphs*. Technical Report AHPCC-TR-99-013. University of New Mexico: Department of Electrical and Computer Engineering, Sept. 1999.

[Ban+08] M. Banikazemi, J. Hafner, W. Belluomini, K. Rao, D. Poff, and B. Abali. „Flipstone: Managing Storage with Fail-in-place and Deferred Maintenance Service Models“. In: *SIGOPS Oper. Syst. Rev.* 42.1 (Jan. 2008), pp. 54–62.

[BDM15] K. Brown, J. Domke, and S. Matsuoka. „Hardware-Centric Analysis of Network Performance for MPI Applications“. In: *2015 21th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. Melbourne, Australia: IEEE Press, Dec. 2015, p. 8.

[BG07] D. Blake and W. Gore. *Effect of Passive and Active Copper Cable Interconnects on Latency of Infiniband DDR compliant systems.* Sept. 2007.

[BH14] M. Besta and T. Hoefler. „Slim Fly: A Cost Effective Low-diameter Network Topology“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 348–359. ISBN: 978-1-4799-5500-8. DOI: `10.1109/SC.2014.34`.

[Bir+15] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. „Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics“. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI)*. Santa Clara, CA: IEEE, Aug. 2015, pp. 1–9. DOI: `10.1109/HOTI.2015.22`.

[BM06] T. Bjerregaard and S. Mahadevan. „A Survey of Research and Practices of Network-on-chip“. In: *ACM Comput. Surv.* 38.1 (June 2006). ISSN: 0360-0300. DOI: `10.1145/1132952.1132953`.

[Bra01] U. Brandes. „A Faster Algorithm for Betweenness Centrality“. In: *Journal of Mathematical Sociology* 25 (2001), pp. 163–177.

[BRM12] R. Birke, G. Rodriguez, and C. Minkenberg. „Towards Massively Parallel Simulations of Massively Parallel High-performance Computing Systems“. In: *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*. SIMUTOOLS '12. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012, pp. 291–298. ISBN: 978-1-4503-1510-4. URL: `http://dl.acm.org/citation.cfm?id=2263019.2263065`.

[CBP00] C. D. Carothers, D. Bauer, and S. Pearce. „ROSS: A High-performance, Low Memory, Modular Time Warp System“. In: *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*. PADS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 53–60. ISBN: 0-7695-0667-4. URL: `http://dl.acm.org/citation.cfm?id=336146.336157`.

[CES71] E. G. Coffman Jr., M. Elphick, and A. Shoshani. „System Deadlocks“. In: *ACM Computing Surveys* 3.2 (1971), pp. 67–78. ISSN: 0360-0300. DOI: `10.1145/356586.356588`.

[CGS10] F. Cappello, A. Guermouche, and M. Snir. „On Communication Determinism in Parallel HPC Applications“. In: *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. Aug. 2010, pp. 1–8. DOI: `10.1109/ICCCN.2010.5560143`.

[Che+12] D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker. „The IBM Blue Gene/Q Interconnection Fabric“. In: *IEEE Micro* 32.1 (Jan. 2012), pp. 32–43. ISSN: 0272-1732. DOI: `10.1109/MM.2011.96`.

[Che+16] D. Chen, P. Heidelberger, C. Stunkel, and Y. Sugawara. „An Evaluation of Network Architectures for Next Generation Supercomputers“. In: *Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. PMBS ’16. New York, NY, USA: ACM, 2016, 2:11–2:21. ISBN: 978-1-5090-5218-9. DOI: `10.1109/PMBS.2016.7`.

[Chu+87] F. R. K. Chung, E. G. Coffman Jr., M. I. Reiman, and B. Simon. „The forwarding index of communication networks“. In: *IEEE Transactions on Information Theory* 33.2 (1987), pp. 224–232. ISSN: 0018-9448. DOI: `10.1109/TIT.1987.1057290`.

[CKR96] L. Cherkasova, V. Kotov, and T. Rokicki. „Fibre channel fabrics: evaluation and design“. In: *Proceedings of the 29th Hawaii International Conference on System Sciences*. Vol. 1. Jan. 1996, pp. 53–62. DOI: `10.1109/HICSS.1996.495447`.

[Cop+11] J. Cope, N. Liu, S. Lang, C. D. Carothers, and R. B. Ross. „CODES: Enabling Co-Design of Multi-Layer Exascale Storage Architectures“. In: *Workshop on Emerging Supercomputing Technologies 2011 (WEST 2011)*. Tuscon, Arizona, USA, 2011.

[Cor+01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. 2nd. MIT Press, 2001. ISBN: 978-0-262-03293-3. URL: `http://books.google.com/books?id=NLngYyWFl_YC`.

[Cro+14] C. Croarkin, P. Tobias, J. J. Filliben, B. Hembree, W. Guthrie, L. Trutna, and J. Prins, eds. *NIST/SEMATECH e-Handbook of Statistical Methods*. NIST/SEMATECH, Apr. 2014. URL: `http://www.itl.nist.gov/div898/handbook/`.

[Dal90] W. J. Dally. „Virtual-channel Flow Control“. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA ’90. New York, NY, USA: ACM, 1990, pp. 60–68. ISBN: 0-89791-366-3. DOI: `10.1145/325164.325115`.

[Den+08] W. E. Denzel, J. Li, P. Walker, and Y. Jin. „A Framework for End-to-end Simulation of High-performance Computing Systems“. In: *1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. Simutools ’08. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, 21:1–21:10. ISBN: 978-963-9799-20-2. URL: `http://dl.acm.org/citation.cfm?id=1416222.1416248`.

[DH16] J. Domke and T. Hoefler. „Scheduling-Aware Routing for Supercomputers“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16. Piscataway, NJ, USA: IEEE Press, 2016, 13:1–13:12. ISBN: 978-1-4673-8815-3. URL: `http://dl.acm.org/citation.cfm?id=3014904.3014922`.

[DHM14] J. Domke, T. Hoefler, and S. Matsuoka. „Fail-in-Place Network Design: Interaction between Topology, Routing Algorithm and Failures". In: *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*. SC '14. New Orleans, LA, USA: IEEE Press, Nov. 2014, pp. 597–608. ISBN: 978-1-4799-5500-8. DOI: `10.1109/SC.2014.54`.

[DHM16] J. Domke, T. Hoefler, and S. Matsuoka. „Routing on the Dependency Graph: A New Approach to Deadlock-Free High-Performance Routing". In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '16. New York, NY, USA: ACM, 2016, pp. 3–14. ISBN: 978-1-4503-4314-5. DOI: `10.1145/2907294.2907313`.

[DHN11] J. Domke, T. Hoefler, and W. E. Nagel. „Deadlock-Free Oblivious Routing for Arbitrary Topologies". In: *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Washington, DC, USA: IEEE Computer Society, May 2011, pp. 613–624. ISBN: 0-7695-4385-7.

[Din+13] B. D. d. Dinechin, R. Ayrignac, P. E. Beaucamps, P. Couvert, B. Ganne, P. G. d. Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. „A clustered manycore processor architecture for embedded and accelerated applications". In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2013, pp. 1–6. DOI: `10.1109/HPEC.2013.6670342`.

[Don16] J. Dongarra. *Report on the Sunway TaihuLight System*. Tech Report UT-EECS-16-742. University of Tennessee, June 2016, pp. 1–24. URL: `http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf` (visited on 12/01/2016).

[DS87] W. J. Dally and C. L. Seitz. „Deadlock-Free Message Routing in Multiprocessor Interconnection Networks". In: *IEEE Trans. Comput.* 36.5 (1987), pp. 547–553. ISSN: 0018-9340. DOI: `10.1109/TC.1987.1676939`.

[DT03] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN: 0-12-200751-4.

[DYL02] J. Duato, S. Yalamanchili, and N. Lionel. *Interconnection Networks: An Engineering Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-852-4.

[Faa+12] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. „Cray Cascade: a Scalable HPC System based on a Dragonfly Network". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, 103:1–103:9. ISBN: 978-1-4673-0804-5. URL: `http://dl.acm.org/citation.cfm?id=2388996.2389136`.

[Fle+06] C. Fleiner, R. B. Garner, J. L. Hafner, K. K. Rao, D. R. Kenchammana-Hosekote, W. W. Wilcke, and J. S. Glider. „Reliability of Modular Mesh-connected Intelligent Storage Brick Systems". In: *IBM J. Res. Dev.* 50.2/3 (Mar. 2006), pp. 199–208. ISSN: 0018-8646. DOI: `10.1147/rd.502.0199`.

[Fli+02] J. Flich, P. López, J. C. Sancho, A. Robles, and J. Duato. „Improving InfiniBand Routing Through Multiple Virtual Networks“. In: *Proceedings of the 4th International Symposium on High Performance Computing*. ISHPC '02. London, UK, UK: Springer-Verlag, 2002, pp. 49–63. ISBN: 3-540-43674-X. URL: `http://dl.acm.org/citation.cfm?id=646349.690713`.

[Fli+12] J. Flich, T. Skeie, A. Mejia, O. Lysne, P. López, A. Robles, J. Duato, M. Koibuchi, T. Rokicki, and J. C. Sancho. „A Survey and Evaluation of Topology-Agnostic Deterministic Routing Algorithms“. In: *IEEE Transactions on Parallel and Distributed Systems* 23.3 (Mar. 2012), pp. 405–425. ISSN: 1045-9219. DOI: `10.1109/TPDS.2011.190`.

[Fra+07] P. Francois, O. Bonaventure, B. Decraene, and P. A. Coste. „Avoiding disruptions during maintenance operations on BGP sessions“. In: *IEEE Transactions on Network and Service Management* 4.3 (Dec. 2007), pp. 1–11. ISSN: 1932-4537. DOI: `10.1109/TNSM.2007.021102`.

[Fre77] L. C. Freeman. „A Set of Measures of Centrality Based on Betweenness“. In: *Sociometry* 40.1 (Mar. 1977), pp. 35–41. ISSN: 00380431. URL: `http://www.jstor.org/stable/3033543`.

[FT87] M. L. Fredman and R. E. Tarjan. „Fibonacci heaps and their uses in improved network optimization algorithms“. In: *J. ACM* 34.3 (1987), pp. 596–615. ISSN: 0004-5411. DOI: `10.1145/28869.28874`.

[Gab+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. „Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation“. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[Gar+13] M. García, E. Vallejo, R. Beivide, M. Odriozola, and M. Valero. „Efficient Routing Mechanisms for Dragonfly Networks“. In: *Proceedings of the 2013 42nd International Conference on Parallel Processing*. ICPP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 582–592. ISBN: 978-0-7695-5117-3. DOI: `10.1109/ICPP.2013.72`.

[Glo14] Global Scientific Information and Computing Center. *Failure History of TSUBAME2.0 and TSUBAME2.5*. Apr. 2014. URL: `http://mon.g.gsic.titech.ac.jp/trouble-list/index.htm` (visited on 04/01/2014).

[GR11] E. G. Gran and S.-A. Reinemo. „InfiniBand congestion control: modelling and validation“. In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. SIMUTools '11. ICST, Brussels, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pp. 390–397. ISBN: 978-1-936968-00-8. URL: `http://dl.acm.org/citation.cfm?id=2151054.2151122`.

[GSI12] GSIC, Tokyo Institute of Technology. *Tsubame2.0 Architecture & Services*. Nov. 2012. URL: `http://www.gsic.titech.ac.jp/sites/default/files/0-tsubame.pdf`.

[Hat11] T. Hatazaki. „Tsubame-2 - a 2.4 PFLOPS peak performance system“. In: *2011 Optical Fiber Communication Conference and Exhibition/National Fiber Optic Engineers Conference (OFC/NFOEC)*. Mar. 2011, pp. 1–3.

[Hen95] R. L. Henderson. „Job Scheduling Under the Portable Batch System“. In: *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. IPPS ’95. London, UK, UK: Springer-Verlag, 1995, pp. 279–294. ISBN: 3-540-60153-8. URL: `http://dl.acm.org/citation.cfm?id=646376.689372`.

[HL08] L. Hsu and C. Lin. *Graph Theory and Interconnection Networks*. Taylor & Francis, 2008. ISBN: 978-1-4200-4482-9. URL: `http://books.google.com/books?id=vbxdqhDKOSYC`.

[HMS89] M. C. Heydemann, J. Meyer, and D. Sotteau. „On Forwarding Indices of Networks“. In: *Discrete Appl. Math.* 23.2 (May 1989), pp. 103–123. ISSN: 0166-218X. DOI: `10.1016/0166-218X(89)90022-X`.

[HR82] G. Ho and C. Ramamoorthy. „Protocols for Deadlock Detection in Distributed Database Systems“. In: *IEEE Transactions on Software Engineering* SE-8.6 (1982), pp. 554–557. ISSN: 0098-5589. DOI: `10.1109/TSE.1982.235884`.

[HS11] T. Hoefler and M. Snir. „Generic Topology Mapping Strategies for Large-scale Parallel Architectures“. In: *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS’11)*. Tucson, AZ: ACM, June 2011, pp. 75–85. ISBN: 978-1-4503-0102-2.

[HS97] G. Hahn and G. Sabidussi. *Graph Symmetry: Algebraic Methods and Applications*. NATO ASI series / C: NATO ASI series. Springer, 1997. ISBN: 978-0-7923-4668-5. URL: `http://books.google.com/books?id=-tIaXdII8egC`.

[HSL08] T. Hoefler, T. Schneider, and A. Lumsdaine. „Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks“. In: *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. ISBN: 978-1-4244-2640.

[HSL09] T. Hoefler, T. Schneider, and A. Lumsdaine. „Optimized Routing for Large-Scale InfiniBand Networks“. In: *17th Annual IEEE Symposium on High Performance Interconnects (HOTI 2009)*. Aug. 2009.

[IEE16] IEEE Standards Association. „IEEE Standard for Ethernet“. In: *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)* (Mar. 2016), pp. 1–4017. DOI: `10.1109/IEEESTD.2016.7428776`.

[Inf10] InfiniBand® Trade Association. *Supplement to InfiniBandTM Architecture Specification Volume 1 Release 1.2.1 (Annex A16: RDMA over Converged Ethernet (RoCE))*. Apr. 2010.

[Inf12] InfiniBand® Trade Association. *InfiniBandTM Architecture Specification Volume 2 Release 1.3 (Physical Specification)*. Nov. 2012.

[Inf15] InfiniBand® Trade Association. *InfiniBandTM Architecture Specification Volume 1 Release 1.3 (General Specifications)*. Mar. 2015.

[Ino10] T. Inoue. *The 6D Mesh/Torus Interconnect of K Computer*. Slides. Nov. 2010. URL: `http://www.fujitsu.com/downloads/TC/sc10/interconnect-of-k-computer.pdf`.

[Jai+16] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale. „Evaluating HPC Networks via Simulation of Parallel Workloads“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16. Piscataway, NJ, USA: IEEE Press, 2016, 14:1–14:12. ISBN: 978-1-4673-8815-3. URL: `http://dl.acm.org/citation.cfm?id=3014904.3014923`.

[Jia+13] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim. „A detailed and flexible cycle-accurate Network-on-Chip simulator“. In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 86–96. DOI: `10.1109/ISPASS.2013.6557149`.

[Joh75] D. B. Johnson. „Priority Queues with Update and Finding Minimum Spanning Trees“. In: *Information Processing Letters* 4.3 (Sept. 1975), pp. 53–57. DOI: `10.1016/0020-0190(75)90001-0`.

[Jok+15] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta. „Quiet Neighborhoods: Key to Protect Job Performance Predictability“. In: *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Hyderabad: IEEE, May 2015, pp. 449–459. DOI: `10.1109/IPDPS.2015.87`.

[Juc08] G. Juckeland. *Introduction to High Performance Computing at ZIH, Architecture of the PC Farm (Deimos)*. Oct. 2008. URL: `http://wwwpub.zih.tu-dresden.de/~mlieber/slides/Architecture.pdf` (visited on 12/01/2016).

[KBB08] P. Kogge, K. Bergman, and S. Borkar. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. Tech. rep. TR-2008-13. Department of Computer Science and Engineering, Notre Dame, Indiana: University of Notre Dame, Sept. 2008.

[KBD07] J. Kim, J. Balfour, and W. Dally. „Flattened Butterfly Topology for On-Chip Networks“. In: *Computer Architecture Letters* 6.2 (Feb. 2007), pp. 37–40. ISSN: 1556-6056. DOI: `10.1109/L-CA.2007.10`.

[Kim+08] J. Kim, W. J. Dally, S. Scott, and D. Abts. „Technology-Driven, Highly-Scalable Dragonfly Topology“. In: *ACM SIGARCH Comput. Architecture News* 36.3 (June 2008), pp. 77–88. ISSN: 0163-5964. DOI: `10.1145/1394608.1382129`.

[Kin+09] M. A. Kinsy, M. H. Cho, T. Wen, E. Suh, M. van Dijk, and S. Devadas. „Application-aware deadlock-free oblivious routing“. In: *Proceedings of the 36th annual International Symposium on Computer Architecture*. ISCA '09. New York, NY, USA: ACM, 2009, pp. 208–219. ISBN: 978-1-60558-526-0. DOI: `10.1145/1555754.1555782`.

[KK98] G. Karypis and V. Kumar. „Multilevel K-way Partitioning Scheme for Irregular Graphs“. In: *J. Parallel Distrib. Comput.* 48.1 (Jan. 1998), pp. 96–129. ISSN: 0743-7315. DOI: `10.1006/jpdc.1997.1404`.

[KKI99] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. „Failure Data Analysis of a LAN of Windows NT Based Computers“. In: *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*. SRDS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 178–. ISBN: 0-7695-0290-3. URL: `http://dl.acm.org/citation.cfm?id=829524.831049`.

[Koi+01] M. Koibuchi, A. Funahashi, A. Jouraku, and H. Amano. „L-turn routing: an adaptive routing in irregular networks“. In: *International Conference on Parallel Processing*. Sept. 2001, pp. 383–392. DOI: `10.1109/ICPP.2001.952084`.

[Koi+03] M. Koibuchi, A. Jouraku, K. Watanabe, and H. Amano. „Descending layers routing: a deadlock-free deterministic routing using virtual channels in system area networks with irregular topologies“. In: *2003 International Conference on Parallel Processing*. Oct. 2003, pp. 527–536. DOI: `10.1109/ICPP.2003.1240620`.

[Koi+12] M. Koibuchi, H. Matsutani, H. Amano, D. F. Hsu, and H. Casanova. „A case for random shortcut topologies for HPC interconnects“. In: *SIGARCH Comput. Archit. News* 40.3 (June 2012), pp. 177–188. ISSN: 0163-5964. DOI: `10.1145/2366231.2337179`.

[Leó+16] E. A. León, I. Karlin, A. Bhatele, S. H. Langer, C. Chambreau, L. H. Howell, T. D’Hooge, and M. L. Leininger. „Characterizing Parallel Scientific Applications on Commodity Clusters: An Empirical Study of a Tapered Fat-tree“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16. Piscataway, NJ, USA: IEEE Press, 2016, 78:1–78:12. ISBN: 978-1-4673-8815-3. URL: `http://dl.acm.org/citation.cfm?id=3014904.3015009`.

[LFD01] P. López, J. Flich, and J. Duato. „Deadlock-Free Routing in InfiniBand through Destination Renaming“. In: *Proceedings of the 2001 International Conference on Parallel Processing*. ICPP ’02. Washington, DC, USA: IEEE Computer Society, 2001, pp. 427–436. ISBN: 0-7695-1257-7. URL: `http://dl.acm.org/citation.cfm?id=645535.656990`.

[Liu+13] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. „F10: A Fault-tolerant Engineered Network“. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. NSDI ’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 399–412. URL: `http://dl.acm.org/citation.cfm?id=2482626.2482665`.

[Liu+15] N. Liu, A. Haider, X.-H. Sun, and D. Jin. „FatTreeSim: Modeling Large-scale Fat-Tree Networks for HPC Systems and Data Centers Using Parallel and Discrete Event Simulation“. In: *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. SIGSIM PADS ’15. New York, NY, USA: ACM, 2015, pp. 199–210. ISBN: 978-1-4503-3583-6. DOI: `10.1145/2769458.2769474`.

[LLS04] D. Li, X. Lu, and J. Su. „Graph-Theoretic Analysis of Kautz Topology and DHT Schemes“. In: *NPC*. 2004, pp. 308–315.

[Los14] Los Alamos National Laboratory. *Operational Data to Support and Enable Computer Science Research*. Apr. 2014. URL: `https://institute.lanl.gov/data/fdata/` (visited on 04/01/2014).

[LS01] O. Lysne and T. Skeie. „Load Balancing of Irregular System Area Networks Through Multiple Roots“. In: *Proceedings of the International Conference on Communication in Computing, CIC 2001*. CSREA Press, 2001, pp. 165–171.

[LST14] D. H. Larkin, S. Sen, and R. E. Tarjan. „A Back-to-Basics Empirical Study of Priority Queues“. In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, Jan. 2014, pp. 61–72. URL: `http://dl.acm.org/citation.cfm?id=2790174.2790181`.

[Luc+14] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist, G. Grider, R. Haring, J. Hittinger, A. Hoisie, D. Klein, P. Kogge, R. Lethin, V. Sarkar, R. S. Schreiber, J. Shalf, T. Sterling, and R. Stevens. *DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges*. Technical Report. USDOE Office of Science (SC)(United States), Feb. 2014.

[Lys+06] O. Lysne, T. Skeie, S.-A. Reinemo, and I. Theiss. „Layered routing in irregular networks“. In: *IEEE Transactions on Parallel and Distributed Systems* 17 (2006), pp. 51–65.

[Man+11] K. L. Man, K. Yedluri, H. K. Kapoor, C.-U. Lei, E. G. Lim, and J. M. „Highly Resilient Minimal Path Routing Algorithm for Fault Tolerant Network-on-Chips“. In: *Procedia Engineering* 15 (2011). {CEIS} 2011, pp. 3406–3410. ISSN: 1877-7058. DOI: `10.1016/j.proeng.2011.08.638`.

[McC+15] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. „Efficient Synthesis of Network Updates“. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 196–207. ISBN: 978-1-4503-3468-6. DOI: `10.1145/2737924.2737980`.

[Mej+06] A. Mejia, J. Flich, J. Duato, S.-A. Reinemo, and T. Skeie. „Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori“. In: *20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006*. 2006, p. 10. DOI: `10.1109/IPDPS.2006.1639341`.

[Mel13] Mellanox Technologies. *Mellanox OFED for Linux User Manual Rev. 2.0-3.0.0*. Aug. 2013. URL: `http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.0-3.0.0.pdf` (visited on 03/16/2016).

[Mel16] Mellanox Technologies. *Product Brief: Unified Fabric Manager Software*. Mar. 2016. URL: `http://www.mellanox.com/related-docs/prod_management_software/PB_UFM_InfiniBand.pdf` (visited on 03/16/2016).

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. June 2015. URL: `http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf` (visited on 03/16/2016).

[Mic15] Microsoft Corporation. *Whitepaper: Microsoft's Cloud Networks*. Tech. rep. 2015. URL: `http://download.microsoft.com/download/9/9/A/99ADCE75-5F63-4E47-905C-F511EE7D3786/Microsofts_Cloud_Networks_Strategy_Brief.pdf` (visited on 01/15/2017).

[MN98] M. Matsumoto and T. Nishimura. „Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator“. In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: `10.1145/272991.272995`.

[Mub+14] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. „A Case Study in Using Massively Parallel Simulation for Extreme-scale Torus Network Codesign“. In: *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. SIGSIM PADS '14. New York, NY, USA: ACM, 2014, pp. 27–38. ISBN: 978-1-4503-2794-7. DOI: `10.1145/2601381.2601383`.

[Nel04] W. Nelson. *Applied Life Data Analysis*. Wiley Series in Probability and Statistics. Wiley, 2004. ISBN: 978-0-471-64462-0. URL: `http://books.google.com/books?id=qyyjJIL40S0C`.

[Oak14] Oak Ridge National Laboratory. *Summit, Oak Ridge's next High Performance Supercomputer*. Nov. 2014. URL: `https://www.olcf.ornl.gov/summit/` (visited on 12/01/2016).

[OGP03] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. „Why do internet services fail, and what can be done about it?“ In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*. Vol. 4. USITS '03. Berkeley, CA, USA: USENIX Association, 2003. URL: `http://dl.acm.org/citation.cfm?id=1251460.1251461`.

[Öhr+95] S. R. Öhring, M. Ibel, S. K. Das, and M. J. Kumar. „On generalized fat trees“. In: *IPPS ’95: Proceedings of the 9th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1995, p. 37. ISBN: 0-8186-7074-6.

[OL05] E. Okorafor and M. Lu. „Percolation routing in a three-dimensional multicomputer network topology using optical interconnection“. In: *J. Opt. Netw.* 4.3 (Mar. 2005), pp. 157–175. DOI: `10.1364/JON.4.000157`.

[Olo16] A. Olofsson. *Whitepaper: Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip*. Tech. rep. 2016. URL: `https://www.parallella.org/docs/e5_1024core_soc.pdf` (visited on 01/15/2017).

[Ope07] OpenSim Ltd. *InfiniBand model from Mellanox*. July 2007. URL: `https://omnetpp.org/9-articles/software/3473--sp-364490752` (visited on 12/01/2016).

[Pal+06] M. Palesi, R. Holsmark, S. Kumar, and V. Catania. „A Methodology for Design of Application Specific Deadlock-free Routing Algorithms for NoC Systems“. In: *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’06. New York, NY, USA: ACM, 2006, pp. 142–147. ISBN: 1-59593-370-0. DOI: `10.1145/1176254.1176289`.

[Pan+14] Z. Pang, M. Xie, J. Zhang, Y. Zheng, G. Wang, D. Dong, and G. Suo. „The TH Express high performance interconnect networks“. In: *Frontiers of Computer Science* 8.3 (2014), pp. 357–366. ISSN: 2095-2236. DOI: `10.1007/s11704-014-3500-9`.

[Pfi02] G. F. Pfister. „An Introduction to the InfiniBand Architecture“. In: *High Performance Mass Storage and Parallel I/O:Technologies and Applications*. Wiley-IEEE Press, 2002, pp. 616–632. ISBN: 978-0-470-54483-9. URL: `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5264600`.

[Pri+13a] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler. „Bandwidth-optimal all-to-all exchanges in fat tree networks“. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS ’13. New York, NY, USA: ACM, 2013, pp. 139–148. ISBN: 978-1-4503-2130-3. DOI: `10.1145/2464996.2465434`.

[Pri+13b] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler. „Fast Pattern-Specific Routing for Fat Tree Networks“. In: *ACM Trans. Archit. Code Optim.* 10.4 (Dec. 2013), 36:1–36:25. ISSN: 1544-3566. DOI: `10.1145/2555289.2555293`.

[Pue+99] V. Puente, R. Beivide, J. A. Gregorio, J. M. Prellezo, J. Duato, and C. Izu. „Adaptive Bubble Router: A Design to Improve Performance in Torus Networks“. In: *Proceedings of the 1999 International Conference on Parallel Processing*. ICPP ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 58–67. ISBN: 0-7695-0350-0. URL: `http://dl.acm.org/citation.cfm?id=850940.852882`.

[PV97] F. Petrini and M. Vanneschi. „k-ary n-trees: high performance networks for massively parallel architectures“. In: *11th International Parallel Processing Symposium*. Apr. 1997, pp. 87–93. DOI: `10.1109/IPPS.1997.580853`.

[Rei+11] S.-A. Reinemo, E. G. Gran, T. Skeie, and O. Lysne. *InfiniBand Congestion Control*. Published: Contributed talk at the 2011 OpenFabrics International Workshop, Monterey, USA. Apr. 2011.

[Rei+12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. „Abstractions for Network Update“. In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’12. New York, NY, USA: ACM, 2012, pp. 323–334. ISBN: 978-1-4503-1419-0. DOI: `10.1145/2342356.2342427`.

[RFR09] A. Rezazadeh, M. Fathy, and G. Rahnavard. „Evaluating Throughput of a Wormhole-Switched Routing Algorithm in NoC with Faults“. In: *International Conference on Network and Service Security*. N2S ’09. June 2009, pp. 1–5.

[RL08] R. Ramanujam and B. Lin. „Near-optimal oblivious routing on three-dimensional mesh networks“. In: *IEEE International Conference on Computer Design*. ICCD 2008. 2008, pp. 134–141. DOI: `10.1109/ICCD.2008.4751852`.

[Rod+09] G. Rodriguez, C. Minkenberg, R. Beivide, R. P. Luijten, J. Labarta, and M. Valero. „Oblivious routing schemes in extended generalized Fat Tree networks“. In: *IEEE International Conference on Cluster Computing and Workshops, 2009 (CLUSTER ’09)*. Aug. 2009, pp. 1–8. DOI: `10.1109/CLUSTR.2009.5289145`.

[RR10] T. Rauber and G. Rünger. *Parallel Programming - for Multicore and Cluster Systems*. Springer, 2010. ISBN: 978-3-642-04817-3.

[RZC11] S. Raza, Y. Zhu, and C.-N. Chuah. „Graceful Network State Migrations“. In: *IEEE/ACM Trans. Netw.* 19.4 (Aug. 2011), pp. 1097–1110. ISSN: 1063-6692. DOI: `10.1109/TNET.2010.2097604`.

[San+02] J. C. Sancho, J. Flich, A. Robles, P. López, and J. Duato. „Analyzing the Influence of Virtual Lanes on the Performance of InfiniBand Networks“. In: *IPDPS ’02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2002, p. 72. ISBN: 0-7695-1573-8.

[SB02] S. Sharma and P. K. Bansal. „A new fault tolerant multistage interconnection network“. In: *2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering*. Vol. 1. TENCON ’02. Oct. 2002, pp. 347–350. DOI: `10.1109/TENCON.2002.1181285`.

[SBH16] T. Schneider, O. Bibartiu, and T. Hoefler. „Ensuring Deadlock-Freedom in Low-Diameter InfiniBand Networks“. In: *24th IEEE Annual Symposium on High-Performance Interconnects*. HOTI’16. Santa Clara, CA, USA, Aug. 2016, pp. 1–8. DOI: `10.1109/HOTI.2016.015`.

[Sch+91] M. D. Schroeder, A. Birell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. „Autonet: A High-speed, Self-Configuring Local Area Network Using Point-to-Point Links“. In: *IEEE Journal on Selected Areas in Communications* 9.8 (Oct. 1991).

[SG10] B. Schroeder and G. Gibson. „A Large-Scale Study of Failures in High-Performance Computing Systems“. In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (Dec. 2010), pp. 337–350. ISSN: 1545-5971. DOI: `10.1109/TDSC.2009.4`.

[Shi+09] K. S. Shim, M. H. Cho, M. Kinsy, T. Wen, M. Lis, G. E. Suh, and S. Devadas. „Static virtual channel allocation in oblivious routing“. In: *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. NOCS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 38–43. ISBN: 978-1-4244-4142-6. DOI: `10.1109/NOCS.2009.5071443`.

[Sin+12] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. „Jellyfish: Networking Data Centers Randomly". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 225–238. ISBN: 978-931971-92-8. URL: `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/singla`.

[SKC14] M. G. Solomon, D. Kim, and J. L. Carrell. *Fundamentals Of Communications And Networking*. 2nd. USA: Jones and Bartlett Publishers, Inc., 2014. ISBN: 1-284-06014-4 978-1-284-06014-0. URL: `https://books.google.com/books?id=LVktBAAAQBAJ`.

[Ske+04] T. Skeie, O. Lysne, J. Flich, P. López, A. Robles, and J. Duato. „LASH-TOR: A Generic Transition-Oriented Routing Algorithm". In: *ICPADS '04: Proceedings of the Parallel and Distributed Systems, Tenth International Conference*. Washington, DC, USA: IEEE Computer Society, 2004, p. 595. ISBN: 0-7695-2152-5. DOI: `10.1109/ICPADS.2004.50`.

[SLT02] T. Skeie, O. Lysne, and I. Theiss. „Layered Shortest Path (LASH) Routing in Irregular System Area Networks". In: *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2002, p. 194. ISBN: 0-7695-1573-8.

[Sod+16] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. „Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36.2 (Mar. 2016), pp. 34–46. ISSN: 0272-1732. DOI: `10.1109/MM.2016.25`.

[SRD00a] J. C. Sancho, A. Robles, and J. Duato. „A new methodology to compute deadlock-free routing tables for irregular networks". In: *Network-Based Parallel Computing. Communication, Architecture, and Applications. 4th International Workshop, CANPC 2000. Proceedings (Lecture Notes in Computer Science Vol.1797)*. Berlin, Germany, 2000, pp. 45–60.

[SRD00b] J. C. Sancho, A. Robles, and J. Duato. „A Flexible Routing Scheme for Networks of Workstations". In: *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*. London, UK: Springer-Verlag, 2000, pp. 260–267. ISBN: 3-540-41128-3.

[SRD04] J. C. Sancho, A. Robles, and J. Duato. „An Effective Methodology to Improve the Performance of the Up*/down* Routing Algorithm". In: *IEEE Transactions on Parallel and Distributed Systems* 15.8 (2004), pp. 740–754.

[Str+16] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. *TOP500*. Nov. 2016. URL: `http://www.top500.org/` (visited on 12/01/2016).

[Sub+12] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda. „Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, 70:1–70:12. ISBN: 978-1-4673-0804-5. URL: `http://dl.acm.org/citation.cfm?id=2388996.2389091`.

[Suh+95] Y.-J. Suh, B. V. Dao, J. Duato, and S. Yalamanchili. „Software based fault-tolerant oblivious routing in pipelined networks". In: *Proceedings of the 1995 International Conference on Parallel Processing. August 1995, I 101 - I*. Aug. 1995, p. 105.

[SWM03] T. Shanley, J. Winkles, and I. MindShare. *InfiniBand Network Architecture*. PC System Architecture Series. Pearson Addison Wesley Prof, 2003. ISBN: 978-0-321-11765-6. URL: `http://books.google.com/books?id=4s0BNIxPsdIC`.

[Szl06] E. Szlachcic. „Fault Tolerant Topological Design for Computer Networks“. In: *International Conference on Dependability of Computer Systems*. DepCos-RELCOMEX ’06. May 2006, pp. 150–159. DOI: `10.1109/DEPCOS-RELCOMEX.2006.25`.

[Tou80] S. Toueg. „Deadlock- and livelock-free packet switching networks“. In: *STOC ’80: Proceedings of the 12th annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1980, pp. 94–99. ISBN: 0-89791-017-6. DOI: `10.1145/800141.804656`.

[Trä13] J. L. Träff. „A Note on (Parallel) Depth- and Breadth-First Search by Arc Elimination“. In: *CoRR* abs/1305.1222 (2013). URL: `http://dblp.uni-trier.de/db/journals/corr/corr1305.html#abs-1305-1222`.

[VAK10] A. Verma, S. Ajit, and D. Karanki. *Reliability and Safety Engineering*. Springer Series in Reliability Engineering. Springer, 2010. ISBN: 978-1-84996-231-5. URL: `https://books.google.com/books?id=OgWhCgAAQBAJ`.

[VH08] A. Varga and R. Hornig. „An overview of the OMNeT++ simulation environment“. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. Simutools ’08. ICST, Brussels, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, 60:1–60:10. ISBN: 978-963-9799-20-2. URL: `http://dl.acm.org/citation.cfm?id=1416222.1416290`.

[VSR15] J. C. Villanueva, T. Skeie, and S.-A. Reinemo. *Whitepaper: Routing and Fault-Tolerance Capabilities of the Fabriscale FM compared to OpenSM*. Tech. rep. Fabriscale, July 2015, p. 10. URL: `http://fabriscale.com/wp-content/uploads/2015/07/whitepaper_isc2015.pdf` (visited on 03/16/2016).

[WC04] G. Wang and J. Chen. „On fault tolerance of 3-dimensional mesh networks“. In: *7th International Symposium on Parallel Architectures, Algorithms and Networks*. May 2004, pp. 149–154. DOI: `10.1109/ISPAN.2004.1300473`.

[WCP13] R. Wang, L. Chen, and T. M. Pinkston. „Bubble Coloring: Avoiding Routing- and Protocol-induced Deadlocks with Minimal Virtual Channel Requirement“. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS ’13. New York, NY, USA: ACM, 2013, pp. 193–202. ISBN: 978-1-4503-2130-3. DOI: `10.1145/2464996.2465436`.

[Wil10] L. J. Wilson. „Managing vendor relations: a case study of two HPC network issues“. In: *Proceedings of the 24th International Conference on Large Installation System Administration*. LISA’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–13. URL: `http://dl.acm.org/citation.cfm?id=1924976.1924991`.

[Wol+16] N. Wolfe, C. D. Carothers, M. Mubarak, R. Ross, and P. Carns. „Modeling a Million-Node Slim Fly Network Using Parallel Discrete-Event Simulation“. In: *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*. SIGSIM-PADS ’16. New York, NY, USA: ACM, 2016, pp. 189–199. ISBN: 978-1-4503-3742-7. DOI: `10.1145/2901378.2901389`.

[Wol+17] N. Wolfe, M. Mubarak, N. Jain, J. Domke, A. Bhatele, C. D. Carothers, and R. B. Ross. „Preliminary Performance Analysis of Multi-rail Fat-tree Networks“. In: *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Short paper; Accepted at CCGrid ’17. Madrid, Spain, May 2017, p. 4.

[Xu10] J. Xu. *Topological Structure and Analysis of Interconnection Networks*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1-4419-5203-9 978-1-4419-5203-5.

[YCM06] H. Yu, I.-H. Chung, and J. Moreira. „Topology Mapping for Blue Gene/L Supercomputer“. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. New York, NY, USA: ACM, 2006. ISBN: 0-7695-2700-0. DOI: `10.1145/1188455.1188576`.

[Yeb+13] P. Yebenes, J. Escudero-Sahuquillo, P. J. Garcia, and F. J. Quiles. „Towards Modeling Interconnection Networks of Exascale Systems with OMNet++“. In: *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. PDP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 203–207. ISBN: 978-0-7695-4939-2. DOI: `10.1109/PDP.2013.36`.

[YHZ09] X. Yu, Z. Huaxin, and S. Zhijun. „Design and implementation of switch module for NS-3“. In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS ’09. ICST, Brussels, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, 3:1–3:10. ISBN: 978-963-9799-70-7. DOI: `10.4108/ICST.VALUETOOLS2009.7658`.

[YJG03] A. B. Yoo, M. A. Jette, and M. Grondona. „SLURM: Simple Linux Utility for Resource Management“. In: *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper*. Ed. by D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4. URL: `http://dx.doi.org/10.1007/10968987_3`.

[Zah+10] E. Zahavi, G. Johnson, D. J. Kerbyson, and M. Lang. „Optimized InfiniBand fat-tree routing for shift all-to-all communication patterns“. In: *Concurr. Comput. : Pract. Exper.* 22.2 (Feb. 2010), pp. 217–231. ISSN: 1532-0626. DOI: `10.1002/cpe.v22:2`.

[Zah10] E. Zahavi. *D-Mod-K Routing Providing Non-Blocking Traffic for Shift Permutations on Real Life Fat Trees*. Tech. rep. Aug. 2010. URL: `https://webee.technion.ac.il/publication-link/index/id/574`.

[Zah12] E. Zahavi. „Fat-tree Routing and Node Ordering Providing Contention Free Traffic for MPI Global Collectives“. In: *J. Parallel Distrib. Comput.* 72.11 (Nov. 2012), pp. 1423–1432. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2012.01.018`.

[ZC12] J. Zhou and Y.-C. Chung. „Tree-turn routing: an efficient deadlock-free routing algorithm for irregular networks“. English. In: *The Journal of Supercomputing* 59.2 (2012), pp. 882–900. ISSN: 0920-8542. DOI: `10.1007/s11227-010-0477-0`.

[Zhe+05] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. „Simulation-based Performance Prediction for Large Parallel Machines“. In: *International Journal of Parallel Programming* 33.2 (June 2005), pp. 183–207. ISSN: 0885-7458. DOI: `10.1007/s10766-005-3582-6`.

[Zhu+15] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. „Congestion Control for Large-Scale RDMA Deployments“. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM ’15. New York, NY, USA: ACM, 2015, pp. 523–536. ISBN: 978-1-4503-3542-3. DOI: `10.1145/2785956.2787484`.

[ZIH13] ZIH, TU Dresden. *Taurus - Network configuration*. Jan. 2013.

[ZL92] X. Zhong and V. M. Lo. „Application-Specific Deadlock Free Wormhole Routing on Multicomputers“. In: *Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*. PARLE ’92. London, UK, UK: Springer-Verlag, 1992, pp. 193–208. ISBN: 3-540-55599-4. URL: `http://dl.acm.org/citation.cfm?id=646421.693827`.