Routing on the Dependency Graph: A New Approach to Deadlock-Free High-Performance Routing

Jens Domke
Institute of Computer Engineering
TU Dresden, Germany
jens.domke@tu-dresden.de

Torsten Hoefler
Computer Science Department
ETH Zurich, Switzerland
htor@inf.ethz.ch

Satoshi Matsuoka
Global Scientific Information and Computing Center
Tokyo Tech, Japan
matsu@is.titech.ac.jp

ABSTRACT

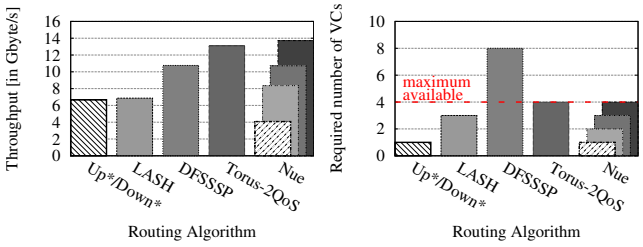
Lossless interconnection networks are omnipresent in high performance computing systems, data centers and network-on-chip architectures. Such networks require efficient and deadlock-free routing functions to utilize the available hardware. Topology-aware routing functions become increasingly inapplicable, due to irregular topologies, which either are irregular by design or as a result of hardware failures. Existing topology-agnostic routing methods either suffer from poor load balancing or are not bounded in the number of virtual channels needed to resolve deadlocks in the routing tables. We propose a novel topology-agnostic routing approach which implicitly avoids deadlocks during the path calculation instead of solving both problems separately. We present a model implementation, called Nue[1], of a destination-based and oblivious routing function. Nue routing heuristically optimizes the load balancing while enforcing deadlock-freedom without exceeding a given number of virtual channels, which we demonstrate based on the InfiniBand architecture.

Keywords

Deadlock-free, destination-based, routing, virtual channels

1. INTRODUCTION

Today's HPC and data center network architectures increasingly embrace functions that enable a tighter interaction between networking hardware and programming models. The best example is Remote Direct Memory Access (RDMA) that enables the programmer to directly instruct the network interface to read and write remote memory. This functionality commonly requires reliable hardware transport between sender and receiver, which can be achieved using iWARP or more recently using lossless Layer 2 network protocols. Such advanced functions have been prevalent in the high-speed networking area, such as InfiniBand [18] or Cray's Cascade [9]. Ethernet was recently extended with Priority Flow Control at Layer 2 to enable the reliable transport needed for RDMA. Indeed, the RoCE protocol enables RDMA for Ethernet.



(a) Throughput for all-to-all **(b) Required VCs (DL-free)**

Figure 1: 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

The downside of lossless Layer 2 networking is that the needed protocols can create deadlock situations [4], where 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. Such deadlocks (DL) can be avoided 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) [26]. Another well-known technique for HPC and on-chip networks (NoC) is the use of virtual channels (VCs, different sets of buffers) to break deadlocks in arbitrary topologies and routing functions [1,30,32]. Dally et al. [6] described this method of conditionally switching between virtual channels along a route, so called virtual channel transition, to obtain a DL-free routing. For example, a k-ary n-cube network needs two VCs if virtual channel transition is possible. However, not all network technologies, e.g., InfiniBand, support this method, in which case the routing functions, such as the layered shortest path routing (LASH) [32], can combine VCs into virtual layers—imagine multiple virtual copies of the actual network—and assign all routes to different layers. Hence, the routing is DL-free, if the combination of routes within each layer is deadlock-free.

Yet, all these concepts have limitations: (1) Topology-aware routings assume perfect topologies and often do not support switch/link failures [7]. (2) Cycle-avoiding routings often cannot balance routes and thus limit global bandwidth [28]. (3) Routings based on virtual channel isolation fail when the required number of VCs is not available [11].

[1] Japanese chimera combining the advantages of existing routings

HPDC'16, May 31–June 04, 2016, Kyoto, Japan.

ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907313

Assume, for example, a 4x4x3 InfiniBand torus network with four terminals per switch and one failed switch (i.e., 47 switches in total) and network support for four VCs. The throughput for various deadlock-free routing strategies, as implemented in InfiniBand's subnet manager, is shown in Fig. 1a, provided that all terminals participate in an all-to-all send operation with 2 KiB messages. Fig. 1b shows the needed number of VCs for these deadlock-free routing algorithms. The topology-aware Torus-2QoS routing [25] 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 [32] are inefficient in comparison to Torus-2QoS. The throughput of the topology-agnostic routing DFSSSP [8] is in-between, however the deadlock-free single-source shortest-path routing (DFSSSP) exceeds the given VC limit and is therefore inapplicable. The achievable throughput while using our new routing approach, Nue, for the 4x4x3 torus is included in Fig. 1a for every number of VCs within the 4-VC limit, showing Nue's resiliency to network faults and ability to offer competitive throughput.

We now describe the underlying idea for Nue routing, i.e., a routing function that overcomes all mentioned limitations (1)–(3), meaning the routing is capable of distributing the paths across the network to increase the throughput. Furthermore, the objectives for our routing function should be to work on arbitrary topologies with all possible numbers of available virtual channels, including network technologies without support for VCs. We assume destination-based routing as is used in many of today's technologies, e.g., InfiniBand.

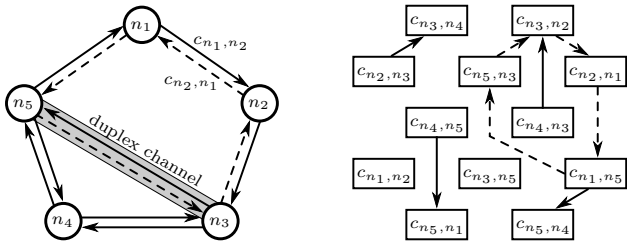
2. DEFINITIONS AND ASSUMPTIONS

We define a network as a multigraph assuming that each pair of network devices can be connected with multiple duplex channels (or links). In this regard, all duplex channels of the interconnection network are logically split into two directed channels of opposite direction, see Fig. 2a. Furthermore, we assume that the channel capacity is uniform and constant over time. If exact device information is required, e.g., for figures, then we use ordered pairs (n_x, n_y) or c_{n_x,n_y}, otherwise a simpler c_i notation is used for channels.

Definition 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. We call $n_x \in N$ a **terminal**, if and only if it exists exactly one n_y with $(n_y, n_x) \in C$, otherwise n_x is called a **switch**.*

Definition 2 (Cycle-free Route)**.** *A **route** (or **path**) P_{n_x,n_y} of length h from node n_x to n_y in $I = G(N, C)$ 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 $1 < q < h$. P_{n_x,n_y} is considered to be **cycle-free**, if from $p \neq q$, with $1 \leq p, q \leq h$ and $c_p = (\cdot, n_u)$, $c_q = (\cdot, n_v)$, it follows that $n_u \neq n_v$. Let ${}^{s}P_{n_x,n_y}$ denote a shortest path from n_x to n_y.*

Definition 3 (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. For multigraphs, we assume 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.*



(a) Ring with shortcut **(b) Corresp. (cyclic) CDG**

Figure 2: Using a shortest-path, counter-clockwise routing for network I (2a) induces the channel dependency graph D (2b); dashed channel dependencies in D form a potential deadlock (induced by paths, with $h = 2$, using dashed channels of I)

Although our definition of the routing function is equivalent to $R' : N \times N \to C$, we will use R to express channel dependencies later on. We consider a routing function for a given network I to be valid, if and only if (iff) the routing has these three properties: *cycle-free*, *destination-based*, and *deadlock-free*, so that the routing is applicable to InfiniBand, Ethernet, and many NoC designs. Especially, the InfiniBand standard demands the use of DL-free routing algorithms, see Section C14-62.1.2 [18]. Dally et al. [6] postulate Theorem 1, which sets the necessary and sufficient condition for the deadlock-freedom of a destination-based routing function.

Theorem 1. *A routing function R for an interconnection network I is deadlock-free if and only if there are no cycles in the corresponding channel dependency graph.*

The channel dependency graph (CDG) is induced by the used routing function R for a given network I as follows:

Definition 4 (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., $(c_p, c_q) \in E$ iff $\exists n_y \in N : R(c_p, n_y) = c_q$.*

Fig. 2b shows the CDG for the network shown in Fig. 2a assuming a shortest-path, counter-clockwise routing function is used (with "shortest-path" as primary path selection criterion). If VCs are supported, but no VC transition, then the alternative way to achieve DL-freedom is the use of virtual channels and to combine them into virtual layers, as mentioned in Section 1. Routes are then assigned to individual layers, so that all routes within one layer induce an acyclic CDG. This technique is used by DFSSSP [8] and LASH [32] routing (among others). However, assuming shortest-path routing, then the minimum number of virtual layers required to achieve deadlock-freedom can exceed the available number of virtual layers, see Section 5.3, and an optimal assignment of routes to layers is an NP-complete problem [8].

Definition 5 (Virtual Layer)**.** *Assume each network channel $c \in C$ can be split into k virtual channels $\{c_1^{virt}, \ldots, c_k^{virt}\}$, with $k \in \mathbb{N}$, then we can determine the i-th **virtual layer** $L_i := G(N, C_i)$ of the interconnection network $I = G(N, C)$ with $C_i := \big\{c_i^{virt} \mid c_i^{virt} \in c \text{ for } c \in C\big\}$. The network I and virtual layer L_i are identical for $k = 1$.*

Our routing algorithm, as we explain hereafter, will use a similar approach, i.e., using virtual layers for deadlock-freedom, but to the best of our knowledge is the first routing function which can be used for every topology and any given number of virtual channels, including $k = 1$.

3. ROUTING IN A DEPENDENCY GRAPH

The current best practice, e.g., as implemented by DFSSSP and LASH, is to decouple the two problems of path creation and deadlock-free assignment to virtual channels. The reason is that both problems require a different graph representation of the network and routes. The deadlock-free assignment needs the CDG, 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. Assume, we can combine the information required to solve both problems within one graph, then we can 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, we can avoid closing cycles in the CDG while calculating the paths instead of breaking the cycles later. Assume, this new graph represents one virtual layer, 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 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 DL-free routes for all source nodes to the destination nodes assigned to this virtual layer. Therefore, all routes in all virtual layers are deadlock-free without exceeding the VC constraint.

4. NUE ROUTING

In the following, we will show how to construct the complete channel dependency graph. Based on this graph, we will develop our deadlock-free, oblivious, and destination-based Nue routing as one example for the general idea of routing within the dependency graph.

4.1 Complete Channel Dependency Graph

To create paths within the CDG, we need a complete representation of all possible channel dependencies, instead of a graph induced by a specific routing. Therefore, we define the complete CDG using the adjacency of channels as follows:

Definition 6 (Complete Channel Dependency Graph)**.** *Let $I = G(N, C)$ be a network according to Definition 1, then the **complete CDG** $\overline{D} := G(C, \overline{E})$, with $\overline{E} \subseteq C \times C$, is defined by $\forall (n_x, n_y), (n_y, n_z) \in C, n_x \neq n_z : \big((n_x, n_y), (n_y, n_z)\big) \in \overline{E}$. We define that the graph $\overline{D}$ is **cycle-free**, if $D \subseteq \overline{D}$ is acyclic, for any CDG D induced by a routing function according to Definition 4. Assuming, the network technology supports $k > 1$ VCs, 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 CDG D induced by a routing function does not have to be stored separately, and can be saved indirectly by assigning

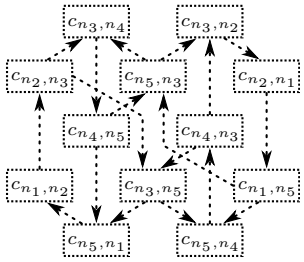


Figure 3: Complete CDG $\overline{D}$ for the 5-ring network with shortcut, see Fig. 2a, assuming $k = 1$; all channels are in *unused* state ($\Rightarrow$ no routing applied, yet)

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. We consider $e \in \overline{E}$ to be *used* iff $e \in E$, i.e., e is induced by a routing R. We mark an edge $e \in \overline{E}$ as *blocked* iff $G(C, E \cup \{e\})$ forms a cyclic graph for an acyclic CDG $D = G(C, E)$. Fig. 3 shows the complete CDG for the 5-node ring network with shortcut, previously shown in Fig. 2a, for $k = 1$. Each vertex/edge of $\overline{D}$ is in the *unused* state, i.e., no routing has been applied.

4.2 Escape Paths

Cherkasova et al. [3] made an important observation: 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 an impasse. Hence, further progress in the algorithm is impossible due to previously added restrictions. While Cherkasova et al. [3] report that this state was observed only rarely for their investigated networks, our experiments show 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" [4, 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.

We adapt the concept of escape paths[2] for our oblivious routing to ensure that at least one valid path—not necessarily shortest—between every given node pair exists, which does not induce a cycle in the CDG. 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. Escape paths inevitably serve as potential imaginary paths which influence the generation and balancing of real paths by Nue. Therefore, the escape paths should induce as few channel dependencies as possible while minimizing the average path length across the escape paths. We are using a spanning tree of I to define the escape paths in $\overline{D}$, since a spanning tree does not induce a cyclic CDG while minimizing the number of channels required to connect all nodes in I.

Definition 7 (Escape Paths)**.** *Let $N^d \subseteq N$ be a set of destination nodes for Nue within the network $I = G(N, C)$ and let $S = G(N, C^s)$, with $C^s \subseteq C$, be a spanning tree*

[2]Not in the sense of a separate set of buffers, but available fall back paths in layer L_i.

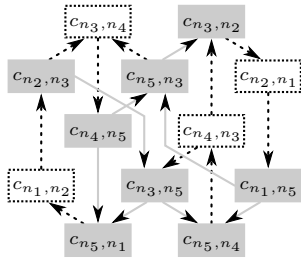
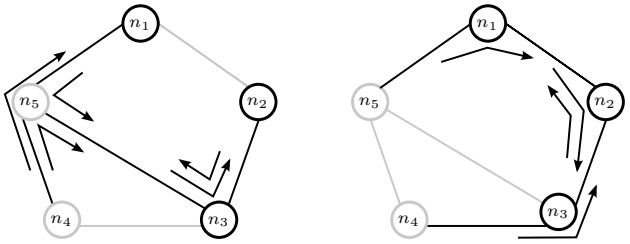


Figure 4: Acyclic escape paths D^s for the 5-node ring network with shortcut, see Fig. 2a, are marked as solid boxes/lines within the complete CDG $\overline{D}$, assuming $k = 1$, destination set $N^d = N$, root node $n_r = n_5$, and spanning tree $S = G(N, C \setminus \{(n_1, n_2), (n_2, n_1), (n_3, n_4), (n_4, n_3)\})$



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

Figure 5: Initial channel dependencies (CDs) of the escape paths shown 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 as black channels)

of network I with root node $n_r \in N$. Then, the ***escape paths*** *$D^s := G(C^s, E^s)$ are a subgraph of $\overline{D}$ induced by a routing $R^s : N \times N^d \to C^s$. Assuming $k > 1$, then D_i^s for virtual layer L_i and root $n_{r,i}$ is defined equivalently, with the substitution of C^s by $C_i^s \subseteq C_i$.*

The term "spanning tree" in this context refers to the originally undirected duplex channels of I, and therefore $(n_x, n_y) \in C^s \implies (n_y, n_x) \in C^s$. Evidently, all cycle-free, destination-based routings R^s induce the same escape paths D^s or D_i^s, respectively, for a given spanning tree. Fig. 4 shows the escape paths marked in the complete CDG for $N^d = N$ when n_5 is used as the root node for S.

The escape paths for our Nue routing have two functions: The first, as mentioned above, is to define an initial set of channel dependencies, which once added cannot be removed to resolve cyclic states in the CDG. Therefore, the escape paths ensure a deadlock-free path from all nodes in N to all nodes in N_i^d within virtual layer L_i. Hence, escape paths for each layer are needed. Despite of having escape paths D_i^s, Nue can end up in an impasse due to the iterative nature of the path creation, as we will see in Sections 4.6.2 and 4.6.3. Therefore, secondly and more importantly, the escape paths are to be actively used by Nue after reaching an unsolvable impasse for a destination node, see Section 4.6.2.

4.3 Choosing Root Node for the Spanning Tree

Nue routing is using escape paths when encountering an impasse, illustrated in Section 4.6.2. Therefore, Nue should ensure that the paths in S are as short as possible. This will reduce latency and oversubscription of individual channels.

Assuming $N_i^d \subset N$, 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. To illustrate this property, we use our previously investigated 5-node ring with shortcut, see Fig. 2a. If n_5 is chosen as root node, since it allows for the shortest average path length in the spanning tree, then the escape paths for $N_i^d = \{n_1, n_2, n_3\}$, as specified in Definition 7, induce five initial channel dependencies. However, if $n_r = n_2$ is chosen instead, then we only have four initial channel dependencies, as shown in Fig. 5.

As a consequence, we propose 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. [12] introduced the metric of betweenness centrality, which is ideal for our purpose to determine the root node for the spanning tree. If we assume a graph $G(N, C)$, then the betweenness centrality $C_B(n)$ for a node $n \in N$ is defined via the absolute number of shortest paths between the two nodes s and t, called σ_{st}, and the number of shortest paths $\sigma_{st}(n)$ which include node n, i.e.,

$$C_B(n) := \sum_{s \neq n \neq t \in N} \frac{\sigma_{st}(n)}{\sigma_{st}}$$

Brandes et al. [2] developed an algorithm to compute the betweenness centrality of every node in the graph. The algorithm for unweighted graphs has a $\mathcal{O}(|N| \cdot |C|)$ time complexity. Unfortunately, the algorithm is not directly applicable to our problem, since we are seeking the most central node with respect to the subset $N_i^d \subset N$ to reduce the initial channel dependencies of the escape paths towards nodes of N_i^d. To adapt our problem, we calculate the convex subgraph for the node set N_i^d, and apply Brandes' algorithm on the convex subgraph.

Definition 8 (Convex Subgraph)**.** *The* ***convex subgraph*** *$H_i := G(N_i^H, C_i^H)$ for a set N_i^d 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, $N_i^H := \{n \in N \mid \forall n_x, n_y \in N_i^d : n = n_x \vee (\cdot, n) \in {}^sP_{n_x,n_y}\}$ and the edge set C_i^H of the convex subgraph is defined analogously.*

The fact that our networks are represented by an unweighted graph allows us to compute the convex subgraph with a time complexity of $\mathcal{O}\big(|N_i^d| \cdot (|N| + |C|)\big)$ with a breadth-first search (forward step) and an inverse traversal of the graph to find all shortest paths (backward step).

After we compute 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 $C_B(n)$ w.r.t. N_i^d. This node will be used as the root node of the spanning tree for the escape paths from all nodes towards the destination nodes in N_i^d. As a remark, for $k = 1$ Nue assigns all destination nodes N_i^d to one virtual layer, i.e., it follows that $H_1 = I$ and Brandes' algorithm can be executed directly on I.

4.4 Dijkstra's Algorithm for the complete CDG

Domke et al. [7, 8] and Hoefler et al. [17] showed the effectiveness, in terms of balancing the paths across the available channels, of a routing algorithm based on Dijkstra's single-source shortest-path algorithm. Their modified Dijkstra algorithm, using a Fibonacci heap to lower the time complexity and extended to work on multigraphs, is combined

Algorithm 1: Dijkstra's Algorithm within $\overline{D}$

Input: $I = G(N, C)$, $\overline{D} = G(C, \overline{E})$, source $n_0 \in N$
Result: P_{n_y,n_0} for all $n_y \in N$ (and $\overline{D}$ is cycle-free)

1 **foreach** *node* $n \in N$ **do**
2 n.distance $\leftarrow \infty$
3 n.usedChannel $\leftarrow \emptyset$
4 **foreach** *channel* $c \in C$ **do**
5 c.distance $\leftarrow \infty$
  /* Need source channel c_0 to start Dijkstra's algorithm: if n_0 is terminal then use unique $(n_0, \cdot)$; if n_0 is switch then $\overline{D}$ has multiple $(n_0, \cdot)$ => $(\emptyset, n_0)$ connects to all */
6 **if** n_0 *is switch* **then** $c_0 \leftarrow (\emptyset, n_0)$ **else** $c_0 \leftarrow (n_0, \cdot)$
7 n_0.distance $\leftarrow 0$
8 c_0.distance $\leftarrow 0$
9 FibonacciHeap $Q \leftarrow \{c_0\}$
10 **while** $Q \neq \emptyset$ **do**
11 $c_p \leftarrow Q$.findMin()
12 **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
13 **if** $c_p.distance + c_q.weight < n_{c_q}.distance$ **then**
14 (c_p, c_q).state $\leftarrow$ used // modifies $\overline{D}$
15 **if** $\overline{D}$ *is cycle-free (see Algorithm 3)* **then**
16 **if** $n_{c_q}.usedChannel \neq \emptyset$ **then**
17 Q.remove(n_{c_q}.usedChannel)
18 Q.add(c_q)
19 c_q.distance $\leftarrow c_p$.distance $+ c_q$.weight
20 n_{c_q}.distance $\leftarrow c_p$.distance $+ c_q$.weight
21 n_{c_q}.usedChannel $\leftarrow c_q$
22 **else**
23 (c_p, c_q).state $\leftarrow$ blocked
  // Optimizations are explained in Sections 4.6.2/4.6.3
24 **if** n_0 *is switch* **then**
25 remove fake channel c_0 from $\overline{D}$ and $(c_0, (n_0, \cdot))$ from $\overline{E}$

with positive weight updates for the used channels after all paths to one destination node are computed.

We use a similar approach, but within the complete CDG $\overline{D}$ or $\overline{D}_i$, respectively. For convenience, we describe Algorithm 1 for $\overline{D}$ only, but it can easily be extended for $\overline{D}_i$. Algorithm 1 computes shortest paths from one source node $n_0 \in N$ to all other nodes in the complete CDG while complying to the cycle-free constraint. Meaning, these paths are not necessarily shortest paths w.r.t. the actual network I. Following the 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 and updates the channel weights similar to DFSSSP [8]. 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 our approach is that channel dependencies are directly considered, see line 15, and routing restrictions can be identified instantaneously, as outlined in Section 3. Therefore, paths do not have to be recomputed to avoid the routing restriction afterwards, as it is the case with smart routing [3], for example.

4.5 Nue Routing Function

Combining the knowledge of Section 4.1–4.4 into one routing function, see Algorithm 2, allows us to achieve our objectives, i.e., to be able to balance the paths globally while not exceeding a given number of VCs, $k \geq 1$, used for deadlock-freedom. In the first step, Nue routing partitions the nodes of the network I into k disjoint subsets, $N_1^d, \ldots, N_k^d$. 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

Algorithm 2: Nue routing calculates all paths within a network I 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 $n_x, n_y \in N$

1 Partition N into k disjoint subsets $N_1^d, \ldots, N_k^d$ of destinations
2 **foreach** *Virtual layer* L_i *with* $i \in \{1, \ldots, k\}$ **do**
  // Check attached comments for details about each step
3 Select a subset of nodes $N_i^d \subseteq N$ for virtual layer L_i
4 Create a convex subgraph H_i for N_i^d // Section 4.3
5 Identify central $n_{r,i} \in N_i^H$ of H_i // Section 4.3
6 Create a new complete CDG $\overline{D}_i$ // Section 4.1
7 Define escape paths D_i^s for root $n_{r,i}$ // Section 4.2
8 **foreach** *Node* $n \in N_i^d$ **do**
9 Identify deadlock-free paths $P_{\cdot,n}$ // Section 4.4
10 Store these paths, e.g., in forwarding tables
11 Update channel weights in $\overline{D}_i$ for these paths

calculate deadlock-free routes for I or not, the partitioning affects the path balancing. Nue routing uses a multilevel k-way partitioning algorithm [19] with $\mathcal{O}(|C|)$ time complexity to partition the network I. Moreover, we implemented a random partitioning and partial clustering, i.e., all terminals connected to a switch are assigned to the same partition. However, Nue with the multilevel k-way partitioning outperformed the other two partitioning algorithms w.r.t. the evaluations carried out in Section 5. 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 this paper and requires further research. For future versions of Nue, we envision improved (optimal) partitioning algorithms that result in an even better path balancing.

The node set N_i^d is used to calculate a convex subgraph H_i. Brandes' algorithm is 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 4.3 for more details. After creating a complete CDG $\overline{D}_i$ for virtual layer L_i, which complies to Definition 6, Nue routing determines the escape paths D_i^s. The acyclic escape paths are derived from a spanning tree rooted at $n_{r,i}$ according to Definition 7, i.e., the channels C_i^s and edges E_i^s 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 our modified Dijkstra algorithm, see Algorithm 1. Each node of N_i^d is used as a source for Algorithm 1. The subsequent weight update for the used channels aims for an improved global balancing of the paths.

4.6 Optimizations for Nue Routing

4.6.1 Numbering of Subgraphs and Cycle Search

Algorithm 1 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. However, Algorithm 1 potentially needs to check $\overline{D}_i$ 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 $\mathcal{O}(|C_i| + |\overline{E}_i|)$.

If we can distinguish between vertex-disjoint, *used* subgraphs of $\overline{D}_i$, induced by a routing R as explained in Section 4.1, then it is possible to avoid a cycle search by applying memorization, since connecting two disjoint, acyclic, and *used* subgraphs with an *used* edge creates a new acyclic subgraph. Therefore, we incorporate an identification number ω for

used and cycle-free subgraphs of $\overline{D}_i$, which is an extension of the three states we utilized before, see Section 4.1. The function $\omega : C_i \cup \overline{E}_i \to \mathbb{Z}_0^+ \cup \{-1\}$, with

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

is used to identify the vertex-disjoint, cycle-free subgraphs and *blocked* edges, i.e., $\omega(e) = -1$. An example is shown in Fig. 6a with $\omega = 1$ pointing to the escape paths of the complete CDG $\overline{D}$, i.e., $\omega(C^s \cup E^s) = 1$ for $D^s = G(C^s, E^s)$ assuming $k = 1$.

The advantage of this is to identify conditions during the routing with 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 and three of them do not require a cycle search:

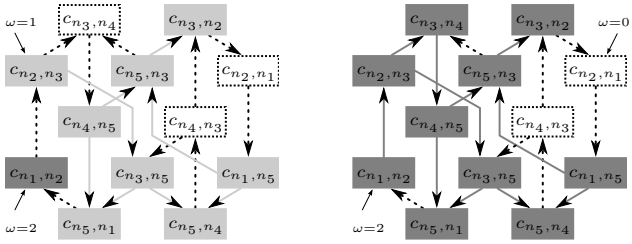
- (a) $\omega(e) = -1 \implies$ no cycle search needed, because the result is known already (these edges are ignored by the conditional loop in line 12 of Algorithm 1);
- (b) $\omega(e) \geq 1 \implies \omega(c_p) = \omega(c_q) = \omega(e) \implies$ no cycle search needed, because e was used before and is therefore part of an acyclic subgraph;
- (c) $\omega(e) = 0 \wedge \omega(c_p) \neq \omega(c_q) \implies$ no cycle search needed, because directed edge e connects two disjoint acyclic subgraphs and therefore cannot close a cycle;
- (d) $\omega(e) = 0 \wedge \omega(c_p) = \omega(c_q) \implies$ cycle search is needed, because e adds an used edge in an acyclic subgraph and might induce a cycle.

Algorithm 3 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 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 it 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.

We will illustrate the conditions (b) to (d) with our previously investigated example of the ring topology with shortcut for $k = 1$. Initially, we assign $\omega = 1$ to the escape paths identifying one cycle-free subgraph. Assume, we start the first routing step with Algorithm 1 at node c_{n_1,n_2} and assign $\omega(c_{n_1,n_2}) = 2$ to it, which will identify the second *used* and cycle-free subgraph of $\overline{D}$, as shown in Fig. 6a. Node c_{n_1,n_2} has only one adjacent node c_{n_2,n_3} available via an *unused* directed edge. Since $\omega(c_{n_1,n_2}) \neq \omega(c_{n_2,n_3})$ we can omit a cycle search, see condition (c). According to lines five to eight of Algorithm 3, both subgraphs, with $\omega = 1$ and $\omega = 2$, are merged into one acyclic subgraph with $\omega = 2$. Now, the adjacent nodes of c_{n_2,n_3} are c_{n_3,n_5} and c_{n_3,n_4} whereby the conditions (b) and (c) apply, respectively. Assuming, Algorithm 1 considers node c_{n_3,n_4} next, then the only available adjacent node is c_{n_4,n_5}, which results in condition (d), where a depth-first search is needed. A DFS from c_{n_4,n_5} for node c_{n_3,n_4} 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. The intermediate state of $\overline{D}$ during Algorithm 1, after these steps have been performed, is shown in Fig. 6b.

Algorithm 3: Search for cyclic *used* subgraphs in $\overline{D}$

```
   Input: D̄ = G(C, Ē), channels c_p, c_q ∈ C
   Result: true if a cycle was found; false otherwise
 1 if ω(c_p, c_q) = −1 then
 2 |   return true          // State described in condition (a)
 3 else if ω(c_p, c_q) ≥ 1 then
 4 |   return false         // State described in condition (b)
 5 else if ω(c_p) ≠ ω(c_q) then
   |   /* Merge two disjoint subgraphs                          */
 6 |   if ω(c_q) = 0 then ω(q) ← ω(c_p) and return false
 7 |   foreach x ∈ C ∪ Ē, with ω(x) = ω(c_q) do
 8 |   |   ω(x) ← ω(c_p)
 9 |   return false         // State described in condition (c)
10 else
   |   /* Perform depth-first search for c_p in subgraph G with
   |      ω(G) = ω(c_p) starting from c_q, see condition (d)   */
11 |   if the D̄.DFS(c_q, ω(c_p)) does not find c_p then
12 |   |   ω(c_p, c_q) ← ω(c_p)
13 |   |   foreach x ∈ C ∪ Ē, with ω(x) = ω(c_q) ≠ 0 do
14 |   |   |   ω(x) ← ω(c_p)
15 |   |   ω(c_q) ← ω(c_p)
16 |   |   return false
17 |   else
18 |   |   ω(c_p, c_q) ← −1
19 |   |   return true
```



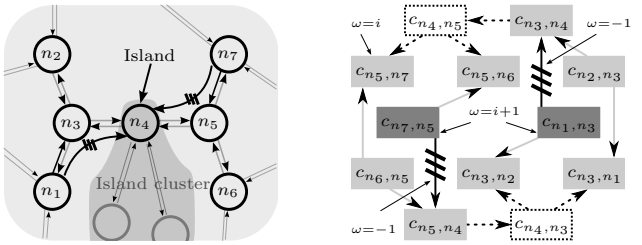
(a) Initial state in Algo. 1 **(b) State after five steps**

Figure 6: State change of $\overline{D}$ after five steps with Algorithm 1, starting from c_{n_1,n_2}; Fig. 6a: initial state (line 14) before the while loop is executed; Fig. 6b: state of $\overline{D}$ after five iterations of the loop

4.6.2 *Solving Impasses for Isolated Nodes/Clusters*

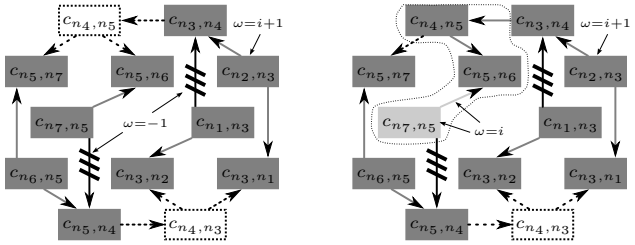
The approach of either randomly removing channel dependencies, as explained in Section 4.2 and mentioned by Cherkasova et al. [3], can lead to impasses during an iterative routing algorithm. Incrementally calculating routes and placing routing restrictions on-demand, as we do, can lead to similar impasses. Meaning, creating isolated parts of the network, we call them *islands*, 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 4.2, for our iterative, destination-based routing function cannot prevent impasses.

To illustrate the problem, we consider a large network I, with a small subnetwork I^* connected as a binary tree, as shown in Fig. 7a, and $k = 1$. The subgraph of the complete CDG $\overline{D}$ for the relevant parts of the network, i.e., for I^*, is shown in Fig. 7b. Assume, our iterative algorithm has calculated all routes for $i - 1$ destinations and is at an intermediate step to calculate the routes towards the i^{th} destination. Therefore, parts of $\overline{D}$ will have $\omega = i$ assigned to it, i.e., $\omega = 1$ for the escape paths plus $i - 1$ destinations. Algorithm 1 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 $(c_{n_1,n_3}, c_{n_3,n_4})$



(a) Subnetwork I^* of I **(b) Intermediate state of $\overline{D}$**

Figure 7: Impasse of Algorithm 1 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 $\overline{D}$



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

Figure 8: The states of the channel dependency graph $\overline{D}$, of the subnetwork I^* of Fig. 7a are shown; Fig. 8a shows the state after solving the island problem for n_4; Fig. 8b highlights the change to $\overline{D}$ when n_4 is used as a shortcut to reach n_5

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 the impasse, we will list three of them in the following. The easiest among them 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 calculated by Algorithm 1. Remember, falling back to the escape path for only a subset of the paths to one destination can violate the destination-based property of Nue. The first option is the least preferred, because impasses happen regularly, hence the escape paths will 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.

We propose to use a local backtracking algorithm, as the third option, whereby we check only the surrounding nodes of distance of 2 hops for alternative routes to the island. This can be accomplished both time- and memory-efficient[3]. 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. So, instead of having Algorithm 1 to overwrite the used channel, see line 21, we store the used channels in a stack. Hence, the stack[4] of valid alternatives, potentially using a longer path, to reach a certain node are stored and is accessible in a backtracking step. Continuing the example from Fig. 7: If Algorithm 1 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. For example, an alternative path to n_3, stored in the stack, is to use channel c_{n_2,n_3}. As Fig. 7b illustrates (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 $\omega = i$. Therefore, using the channel dependency $(c_{n_2,n_3}, c_{n_3,n_4})$ is a valid alternative for our modified Dijkstra algorithm to reach n_4. If multiple valid alternatives exist, then Nue selects the shortest among them—w.r.t. the weight/distance parameters of the channels. After a valid alternative is found for one island node, Algorithm 1 continues to operate as before to ensure that paths into clusters of island nodes are calculated.

4.6.3 Using Formerly Isolated Nodes as Shortcut

In the previous section, we explained how to use a local backtracking to solve routing impasses and how to find paths into island nodes/clusters. Furthermore, these island nodes can be used to shorten the distance to previously discovered nodes. For instance, assume, Algorithm 1 reaches an impasse for the network presented in Fig. 7a and the algorithm cannot find a path to node n_4, as described in Section 4.6.2. 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 1. 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 1 reaches the impasse. The local backtracking algorithm, see Section 4.6.2, enables a valid path to n_4 via n_3. 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, to make use of shortcuts during the routing within the complete CDG, existing channel dependencies have to be considered. While, in theory, it would be possible to invalidate decisions and dependencies for paths which use n_5 as an intermediate node, it will increase the runtime of the routing algorithm. Since the channel dependencies are built incrementally by Algorithm 1, 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, we incorporate the following optimization into Nue: Using islands as shortcuts is only allowed if existing local channel dependencies can be kept in place. This avoids the need to reconsider subsequent dependencies.

We exemplify this algorithm[5] by using the example network of Fig. 7a. Assume, after the impasse of Algorithm 1 and the solution for a path to the island node $n_4 = n_{c_q}$, the current state of $\overline{D}$ is shown in Fig. 8. So, we first have to check whether n_4 can be used as a shortcut to reach n_5 or not, i.e., we 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

[3]The detailed algorithms for the optimizations explained in Sections 4.6.2 and 4.6.3 are not needed to understand the underlaying concept, and are therefore omitted.

[4]For simplicity, the handling of the stack in Algorithm 1 is omitted.

[5]See footnote 3.

intermediate nodes to reach n_6 and subsequent nodes, not shown in Fig. 7a. 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. Our shortcut algorithm 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. 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 n_4 is a valid shortcut for n_5 and 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.

4.7 Correctness, Completeness & Complexity

In the following, we prove that Nue routing is destination-based and cycle-free, see Definition 3, and we prove Nue's deadlock-freedom. Meaning, the CDG D induced by the calculated paths is acyclic, independently of the underlying network or the predefined number of available VCs. Afterwards, we summarize Nue's time and memory complexity.

Lemma 1. *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. Algorithm 1 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 our 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 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 1, 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 n_v is not part of the cycle, 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. □

Lemma 2. *Nue routing is deadlock-free.*

Proof. According to Theorem 1, the routing function is deadlock-free iff the corresponding CDG 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 cycle checks, see Algorithm 1 line 15, and Sections 4.6.2 and 4.6.3, before any of the usedChannel variables are changed, are preventing Nue 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, see Definition 6, and Theorem 1 is applicable. □

Lemma 3. *The Nue routing function ensures connectivity between any pair of two nodes $n_u, n_v \in N$ in the interconnection network $I = G(N, C)$, i.e., $P_{n_u,n_v} \neq \emptyset$.*

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 VC constraint. Therefore, either the network I is disconnected, which contradicts Definition 1, or the variable n_u.usedChannel $= \emptyset$ and the modified Dijkstra algorithm 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 Section 4.6.2. Meaning, if $n_v \in N_i^d$, then from Definition 7 it follows that for every $n_w \in N$ there exists a $(n_w, \cdot) \in C_i^s$ with $R^s\big((n_w, \cdot), n_v\big) \in C^s$, 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^s. This contradicts the initial assumption $P_{n_u,n_v} = \emptyset$, hence Nue routing ensures full connectivity. □

Most terms of the below shown Proposition 1 follow directly from the explanations in previous Sections 4.1–4.6. Therefore, we will focus the following explanations on the most complex and most time consuming term $\mathcal{O}(\ldots)_{\text{Routing}}$, i.e., Algorithm 1 of our Nue routing.

For the 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 $|\overline{E}| \leq \Delta \cdot |C| \leq \Delta^2 \cdot |N|$. The time complexity, excluding the acyclicity check, for our modified Dijkstra algorithm, as presented in Algorithm 1, is $\mathcal{O}(|C_i| \cdot \log |C_i| + |\overline{E}_i|)$ when executed on a complete CDG $\overline{D}_i$. A heap with $\mathcal{O}(1)$ time complexity for the "decrease-key" operation is needed to achieve this complexity. The optimization, see Section 4.6.1 and conditions (a)–(d), results in a differentiated time complexity for the acyclicity check of $\overline{D}_i$, see line 15 of Algorithm 1. In the best case, i.e., condition (a) or (b) are applied, the time complexity is $\mathcal{O}(1)$. The same applies to the "merge" of the two acyclic subgraphs assuming that $\omega(c_q)$ is zero, see line 6 of Algorithm 3. 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 $\mathcal{O}(|N| \cdot (|C_i| + |\overline{E}_i|))$. The time complexity for any depth-first search within a subgraph of $\overline{D_i}$ is $\mathcal{O}(|C_i| + |\overline{E}_i|)$. 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 (d) cannot be applied again.

Proposition 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*

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

while its memory complexity (including storing the result) is

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

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

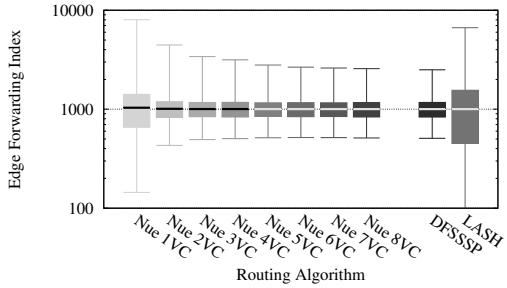


Figure 9: Averaged edge forwarding index metrics: minimum Γ^r_{min} and maximum Γ^r_{max} for the whiskers, and average Γ^r_{avg} ($\pm$ standard deviation Γ^r_{SD}) for the box; for 1,000 random topologies with 125 switches, 1,000 terminals and 1,000 switch-to-switch channels

5. EVALUATION OF NUE ROUTING

We apply Nue to the InfiniBand network technology. We use a publicly available simulation toolchain [7], based on InfiniBand tools and a flit-level simulator written with the OMNeT++ framework. We create different topologies and route these with multiple routing algorithms which are implemented in the production-quality InfiniBand network manager OpenSM version 3.3.16 [25]. The simulator estimates the communication throughput of an all-to-all traffic pattern. Most of the routing algorithms either omit the calculation of switch-to-switch paths entirely, e.g., fat tree routing [33] or Up*/Down* routing [29], or ignore these paths during the DL-avoidance calculation phase, e.g., DFSSSP, since switch-to-switch paths are only used for management messages and not data messages. Therefore, for a fair comparison, we exclude switches from the set of destination nodes for Nue routing while performing the following evaluations.

5.1 Path Length and Edge Forwarding Index

Nue routing does not always create routes along the shortest-paths, depending on the available number of VCs and whether the escape path is used or not. However, non-minimal routes increase latency and may cause higher network congestion. We first analyze the path length of routes created by Nue and compare them to shortest-path algorithms. Additionally, we investigate the edge forwarding index γ for inter-switch ports in the network [15]. This metric allows us to analyze and compare the quality of the routing function in terms of path balancing. A high minimum γ and low maximum γ are indicators for a well balanced routing algorithm.

To evaluate both metrics, we create 1,000 random topologies. Each topology consists of 125 switches interconnected by 1,000 channels, and eight terminals connected to each switch. We apply our Nue routing, with $1 \leq k \leq 8$ virtual channels, as well as LASH and DFSSSP routing, to create DL-free forwarding tables. We collect the following metrics for each topology/routing combination: minimum, maximum, and average edge forwarding index and the standard deviation, i.e., $\gamma^{t,r}_{min}$, $\gamma^{t,r}_{max}$, $\gamma^{t,r}_{avg}$, and $\gamma^{t,r}_{SD}$ for topology t and routing r. These metrics are then arithmetically averaged for all 1,000 topologies: $\Gamma^r_{min} = \frac{\sum_{t=1}^{1,000} \gamma^{t,r}_{min}}{1,000}$ and so forth.

Fig. 9 shows the result as box plots, with whiskers indicating Γ^r_{min} and Γ^r_{max}, and with the box indicating the average Γ^r_{avg} and $\Gamma^r_{avg} \pm \Gamma^r_{SD}$. As we can see, Nue routing performs almost similar to DFSSSP, for $k \geq 4$. It is worth mentioning, that DFSSSP needs at least four VCs to calculate DL-free routes for these topologies, or five VCs in some exceptional cases. LASH's VCs requirement is lower compared to DFSSSP and ranges between two and four. However both, Nue and DFSSSP, clearly outperform LASH w.r.t. the edge forwarding index metric. Even so Fig. 9 indicates that DFSSSP slightly outperforms Nue, we have to keep in mind that Nue routing is designed for arbitrary topologies and supports every given number of VCs. Therefore, Nue will be able to calculate DL-free paths even if we scale up the size of the topologies, while the other routings, such as DFSSSP, will fail due to VC limitations, as we will see in Section 5.3.

The increased γ for $k < 4$ has two reasons: one is the concentration of paths on certain channels to bypass routing restrictions, and the other are longer paths due to the use of the escape paths or parts of them. It is clear that a longer path changes γ for more ports in the network. For all 1,000 random topologies, we measured the maximum path length in the network. For the best case, Nue routing needs only two VCs 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—arithmetic average of 5.3—as DFSSSP, if Nue distributes the paths among at least seven virtual layers. The worst case length of the longest path for Nue is 7–10, depending on the given number of VCs, while it is 6 for DFSSSP/LASH.

The number of fall backs to escape paths depends on many factors, such as topology type, size, number of VCs, and the chosen root node for the spanning tree. For our random topologies with no additional VCs, 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 paper.

5.2 Throughput for (Ir-)regular Topologies

Additionally to the random topologies from Section 5.1, we use the simulation toolchain to measure the throughput for four standard topologies (i.e., fat tree, torus, Kautz graph [23], Dragonfly [20]) and two real-world topologies, namely Cray's Cascade [9] and Tsubame2.5's fat tree [14] (2^{nd} rail; connecting 1,407 compute node). For the Cascade topology, we configured 192 global channels to connect the two Cascade electrical groups. An arbitrary random topology has been chosen among the 1,000 created random topologies for this throughput measurement. All topologies accommodate roughly 1,000 terminals[6] 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 Tab. 1. We assume QDR InfiniBand and a limit of eight VCs for the simulations. The redundancy r listed in Tab. 1 refers to a multiplication of switch-to-switch channels w.r.t. the usual topology definition to increase the port usage.

The flit-level simulator performs an all-to-all send operation with 2 KiB message size between all terminals of the network. An exchange pattern of varying shift distances[7] is used at each terminal to communicate with all other ter-

[6] High memory consumption for the detailed/accurate flit-level simulations prevents us from analysing much larger topologies.

[7] Simulating uniform random injection traffic yields similar behaviour of Nue, and showing these results will not provide further insight.

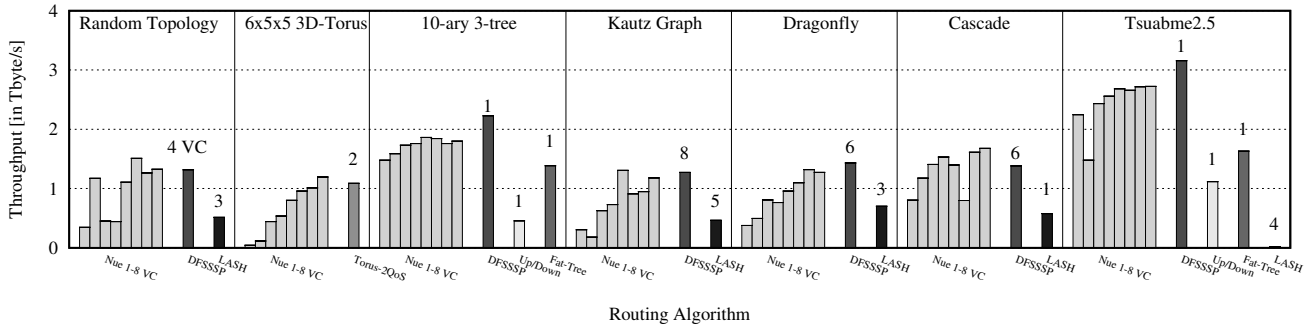


Figure 10: Simulated throughput for all-to-all operation on five standard and two real-world topologies, as configured according to Tab. 1; Nue routing shown for all numbers of VCs between 1 and 8 (from left to right); VC requirement by other routings for deadlock-freedom are shown atop the individual bars

Table 1: Topology configurations (w/ link redundancy r) used for throughput simulations in Fig. 10

Topology	Switches	Terminals	Channels	r
Random	125	1,000	1,000	1
6x5x5 3D-Torus	150	1,050	1,800	4
10-ary 3-tree	300	1,100	2,000	1
Kautz ($d = 7, k = 3$)	150	1,050	1,500	2
Dragonfly ($a = 12$, $p = 6, h = 6, g = 15$)	180	1,080	1,515	1
Cascade (2 groups)	192	1,536	3,072	1
Tsubame2.5	243	1,407	3,384	1

minals [7]. We measure the throughput of all eight routing algorithms that are available in OpenSM version 3.3.16. Impossible topology/routing combinations, such as Torus-2QoS routing for the 10-ary 3-tree, are ignored. We compare Nue routing, for the number of virtual channels $1 \leq k \leq 8$, for a given topology to the other usable algorithms, see Fig. 10.

Besides the simulated throughput for each topology and routing combination, we give the number of needed VCs atop of the bars in Fig. 10. For example, DFSSSP routing needs four VCs for a deadlock-free routing of the random topology. However, DFSSSP usually uses all eight available VCs to optimize the path balancing across the virtual layers, a technique to increase the throughput slightly [5].

We see two trends for all investigated topologies: First, an increase of used VCs for Nue also increases the throughput for the all-to-all communication. This is a result of the decreased $\gamma_{max}^{t,\text{Nue}}$ when we use multiple VCs, as reported in Section 5.1. The outliers from this pattern, e.g., the decrease in throughput for the random topology and four VCs, are correlated 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 VCs was able to route the topology without any fall back. A second trend is, that Nue shows a slight variance in throughput after reaching a certain peak, usually for about $k \geq 5$ in our examples, but this generally depends on topology type and size. We account this behavior 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 [7, 16].

In general, Fig. 10 shows that Nue routing is competitive to the best performing routing for each individual topology, i.e., offers between 83.5% (10-ary 3-tree) and 121.4% (Cascade network) throughput in comparison to Torus-2QoS for the torus and DFSSSP for the other topologies. Occasionally, depending on the given number of VCs, Nue is able to outperform the best competitor. 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 $k \geq 3$ but excluding $k = 6$. Furthermore, given enough VCs, Nue is able to outperform the other routings, such as fat-tree routing or LASH, to a great extent. Therefore, we consider Nue routing to be a adequate alternative to the other investigated algorithms, or at least a suitable fall back in case the best performing algorithm becomes inapplicable, as we will discuss in Section 5.3.

5.3 Runtime and Practical Considerations

In Proposition 1 we mathematically derived the time complexity of Nue routing. To put this into perspective, we compare the runtime of Nue (with 8 VCs) for tori topologies to other deadlock-free routing algorithms implemented in OpenSM. Therefore, we extend the current OpenSM (version 3.3.19) with our Nue routing to achieve a fair comparison and integrate Nue into the toolchain, described in Section 5.

We create 25 3D torus networks with a difference in dimension of at most one, i.e., we start with a grid size for the switches of 2x2x2, 2x2x3, 2x3x3, ..., and go up to 10x10x10. Each of these switches connects to four terminals, hence the 10x10x10 torus accommodates 4,000 terminals. Furthermore, the assumption for this test is that the maximum of available VCs is 8, adjacent switches utilize no channel redundancy, and we randomly inject 1% link/channel failures into the topology. The 1% link failures have been chosen according to the observed annual failure rate of production HPC systems [7]. Besides Nue, we evaluate the runtime of DFSSSP, LASH, and Torus-2QoS routing to calculate deadlock-free routing tables for the same faulty topology. The testbed used is a dual-socket Intel Xeon E5-2620 server with 64 GiB RAM and we pin the InfiniBand fabric simulator (ibsim) to socket 0 and OpenSM to socket 1 to minimize disturbances.

From the results, as shown in Fig. 11, we can draw two main conclusions: First, Nue is competitive in terms of runtime. Nue routing calculates the forwarding tables faster

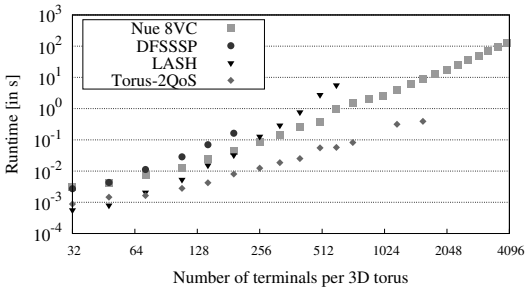


Figure 11: Runtime comparison of DL-free routings for 3D tori (w/o channel redundancy) of various topology sizes with 1% injected link failures; Four terminals per switch (e.g., smallest: 2x2x2 torus with 32 terminals, largest topology: 10x10x10 torus with 4,000 terminals); Missing dots: routing failed

than the topology-agnostic DFSSSP, which has the same time complexity of $\mathcal{O}(|N|^2 \cdot \log|N|)$, see [8]. 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 Fig. 11, either because the algorithms run out of VCs (DFSSSP and LASH exceed the 8 VC limit) or because the failures prevent an analytical solution for the deadlock-free paths problem (Torus-2QoS). Only Nue routing is always applicable while offering good path balancing.

6. RELATED WORK

A well-known example for algorithms avoiding to create cycles in the CDG is the Up*/Down* routing [29]. 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 [28], L-turn routing [22] and segment-based routing (SR) [24] 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 [10] can increase the path balancing. For similar network technologies without virtual channel support the Tree-turn routing [34] or FX [27] routing 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 routing algorithms breaks cycles in the CDG with virtual channels. The destination-based DFSSSP [8] and LASH [32] routings operate similarly in terms of breaking the cycles, i.e., searching for cycles in the CDG and moving individual paths to other virtual layers. Albeit, both algorithms might suffer from a limited number of available VCs. Therefore, LASH-TOR [31] enhanced LASH routing to use Up*/Down* in the last virtual layer if the routes in this layer form an unresolvable cycle. This can result in multiple outgoing ports at a switch for a single destination, hence LASH-TOR is not destination-based in the general case.

Kinsy et al. [21] proposed two application-aware routings, called bandwidth-sensitive oblivious routing (with minimal routes), or BSOR(M). While BSOR operates similar to our approach, meaning its calculates the routes within the CDG, the BSORM routing calculates the routes within the network and breaks cycles afterwards, resembling the method of DFSSSP and LASH. However, the difference between Nue and BSOR is that BSOR randomly deletes edges from the CDG to form an acyclic CDG and solves a multi-commodity flow problem, based on the demands of the application, with a MILP algorithm for small networks. For large networks, BSOR uses Dijkstra's algorithm as a heuristic on a weighted and acyclic CDG for each source/destination pair to balance the application traffic. 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 [3]. The approach of smart routing is to calculate the shortest paths and investigate the induced CDG 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 VCs, the computational cost, which is $\mathcal{O}((\#\text{switch})^9)$, is too high for a practical use in large scale networks.

7. CONCLUSION

The InfiniBand interconnect is currently the #1 network technology—w.r.t. the number of systems in the Top500 list—for high performance computing. Lossless networks, such as InfiniBand or Converged Enhanced Ethernet (CEE), also become more common in data center environments. The main features of these networks are lossless transmission using either credit based flow control in InfiniBand or Priority Flow Control (PFC) with pause frames in Ethernet. Both technologies require deadlock-free routing to function reliably. The same is true for many network-on-chip architectures.

Our approach of applying a graph search algorithm within the complete channel dependency graph instead of the actual network, and our implementation of it, called Nue, is the first reliable strategy to route arbitrary topologies with a limited number of virtual channels, or even in the absence of VCs. Nue routing is tailored for the deadlock-free, oblivious, and destination-based routing needed in CCE 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.

Furthermore, Nue's capability of arbitrarily limiting the number of used VCs allows the combination of DL-freedom and quality of service (QoS), which could be based on the same technology feature. E.g., InfiniBand's service levels are mapped to virtual lanes, which are used by LASH/DFSSSP to avoid deadlocks. So, previously, the choice was either having QoS with topology-aware routing or ignoring QoS and using topology-agnostic routings based on VLs (LASH or DFSSSP). With Nue routing, one could use two VLs for DL-freedom while having four QoS levels, for example.

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 HPC systems, lossless data center fabrics, and NoC architectures. Therefore, we expect a wide adoption of Nue routing and of the concept of routing on the complete channel dependency graph in these fields.

8. REFERENCES

[1] T. Bjerregaard and S. Mahadevan. A Survey of Research and Practices of Network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006.

[2] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[3] L. Cherkasova, V. Kotov, and T. Rokicki. Fibre channel fabrics: evaluation and design. In *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, volume 1, 1996.

[4] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[5] W. J. Dally. Virtual-channel Flow Control. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, New York, USA, 1990. ACM Press.

[6] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36(5):547–553, 1987.

[7] J. Domke, T. Hoefler, and S. Matsuoka. Fail-in-place Network Design: Interaction Between Topology, Routing Algorithm and Failures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, Piscataway, NJ, USA, 2014. IEEE Computer Society.

[8] 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, 2011. IEEE Computer Society.

[9] 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, pages 103:1–103:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[10] 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, 2002. Springer-Verlag.

[11] J. Flich, T. Skeie, A. Mejia, O. Lysne, P. Lopez, A. Robles, J. Duato, M. Koibuchi, T. Rokicki, and J. C. Sancho. A Survey and Evaluation of Topology-Agnostic Deterministic Routing Algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 23(3):405–425, 2012.

[12] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, 1977.

[13] M. Garcia, E. Vallejo, R. Beivide, M. Odriozola, and M. Valero. Efficient Routing Mechanisms for Dragonfly Networks. In *42nd International Conference on Parallel Processing (ICPP)*, 2013.

[14] GSIC. TSUBAME2 Hardware Architecture. http://tsubame.gsic.titech.ac.jp/en/hardware-architecture, Jan. 2016.

[15] M. C. Heydemann, J. Meyer, and D. Sotteau. On Forwarding Indices of Networks. *Discrete Appl. Math.*, 23(2):103–123, May 1989.

[16] 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 Press, 2008.

[17] 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)*, 2009.

[18] InfiniBand Trade Association. *Infiniband Architecture Specification Volume 1, Release 1.2.1*, 2007.

[19] G. Karypis and V. Kumar. Multilevel K-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[20] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 77–88, Washington, DC, USA, 2008. IEEE Computer Society.

[21] 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, 2009. ACM Press.

[22] M. Koibuchi, A. Funahashi, A. Jouraku, and H. Amano. L-turn routing: an adaptive routing in irregular networks. In *International Conference on Parallel Processing*, 2001.

[23] D. Li, X. Lu, and J. Su. Graph-Theoretic Analysis of Kautz Topology and DHT Schemes. In *NPC*, pages 308–315, 2004.

[24] 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 (IPDPS)*, 2006.

[25] Mellanox Technologies. *Mellanox OFED for Linux User Manual*, rev 2.0-3.0.0 edition, 2013.

[26] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer-Verlag, 2013.

[27] 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, 2000. Springer-Verlag.

[28] 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*, volume 1797 of *Lecture Notes in Computer Science*, pages 45–60. Springer Berlin Heidelberg, 2000.

[29] 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. *IEEE Journal on Selected Areas in Communications*, 9(8), 1991.

[30] 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, pages 38–43, Washington, DC, USA, 2009. IEEE Computer Society.

[31] 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 Tenth International Conference on Parallel and Distributed Systems*, Washington, DC, USA, 2004. IEEE Computer Society Press.

[32] 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, 2002. IEEE Computer Society Press.

[33] E. Zahavi, G. Johnson, D. J. Kerbyson, and M. Lang. Optimized InfiniBand fat-tree routing for shift all-to-all communication patterns. *Concurr. Comput. : Pract. Exper.*, 22(2):217–231, 2010.

[34] J. Zhou and Y.-C. Chung. Tree-turn routing: an efficient deadlock-free routing algorithm for irregular networks. *The Journal of Supercomputing*, 59(2):882–900, 2012.