T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

Title	HyperX Topology: First At-Scale Implementation and Comparison to the Fat-Tree
Authors	Jens Domke, Satoshi Matsuoka, Ivan R. Ivanov, Yuki Tsushima, Tomoya Yuki, Akihiro Nomura, Shin ' ichi Miura, Nic McDonald, Dennis L. Floyd, Nicolas Dubé
Citation	SC '19 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, , No. 40,
Pub. date	2019, 11
Note	(C) ACM 2019. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in SC '19 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, http://dx.doi.org/https://doi.org/10.1145/3295500.3356140.

Powered by T2R2 (Science Tokyo Research Repository)

HyperX Topology: First At-Scale Implementation and Comparison to the Fat-Tree

Jens Domke*
Satoshi Matsuoka
RIKEN Center for Computational Science (R-CCS), RIKEN
Kobe, Japan

Ivan R. Ivanov, Yuki Tsushima
Tomoya Yuki, Akihiro Nomura
Shin'ichi Miura
Tokyo Institute of Technology
Tokyo, Japan

Nic McDonald
Dennis L. Floyd
Nicolas Dubé
Hewlett Packard Enterprise (HPE)
Palo Alto, United States of America

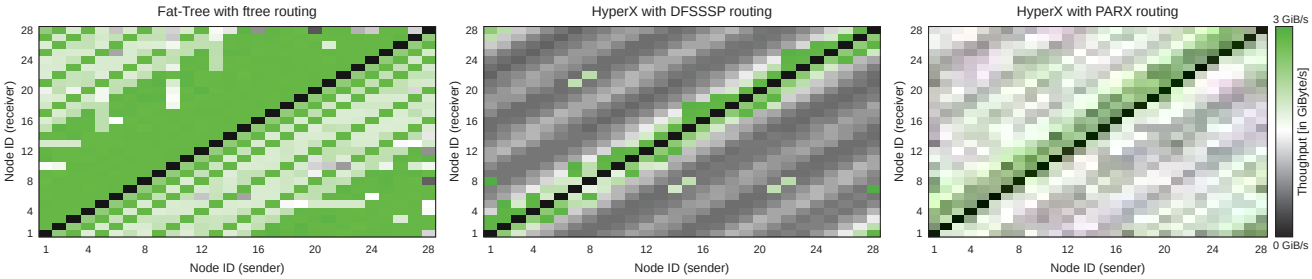


Figure 1: Observable Bandwidth in mpiGraph for 28 Nodes of our Fat-Tree-based and HyperX-based Supercomputer

ABSTRACT

The de-facto standard topology for modern HPC systems and datacenters are Folded Clos networks, commonly known as Fat-Trees. The number of network endpoints in these systems is steadily increasing. The switch radix increase is not keeping up, forcing an increased path length in these multi-level trees that will limit gains for latency-sensitive applications. Additionally, today's Fat-Trees force the extensive use of active optical cables which carries a prohibitive cost-structure at scale. To tackle these issues, researchers proposed various low-diameter topologies, such as Dragonfly. Another novel, but only theoretically studied, option is the HyperX. We built the world's first 3 Pflop/s supercomputer with two separate networks, a 3-level Fat-Tree and a 12x8 HyperX. This dual-plane system allows us to perform a side-by-side comparison using a broad set of benchmarks. We show that the HyperX, together with our novel communication pattern-aware routing, can challenge the performance of, or even outperform, traditional Fat-Trees.

CCS CONCEPTS

• **Computer systems organization → Interconnection architectures**; • **Networks → Network experimentation**; *Network performance analysis*; *Routing protocols*; Packet-switching networks.

*The research was performed while the author was affiliated with GSIC, Tokyo Tech.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC '19, November 17–22, 2019, Denver, CO, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356140

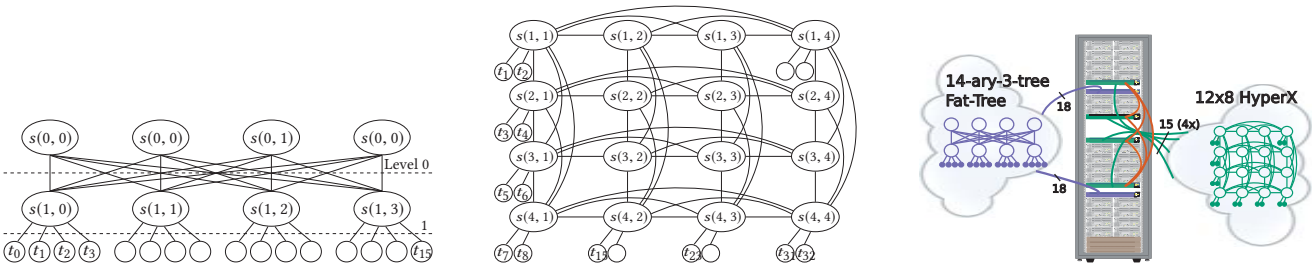
KEYWORDS

HyperX, Fat-Tree, network topology, routing, PARX, InfiniBand

ACM Reference Format:
Jens Domke, Satoshi Matsuoka, Ivan R. Ivanov, Yuki Tsushima, Tomoya Yuki, Akihiro Nomura, Shin'ichi Miura, Nic McDonald, Dennis L. Floyd, and Nicolas Dubé. 2019. HyperX Topology: First At-Scale Implementation and Comparison to the Fat-Tree . In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3295500.3356140

1 INTRODUCTION

The striking scale-out effect, seen in previous generations at the top end of the supercomputer spectrum, e.g., the K computer with over 80,000 compute nodes or the Sunway TaihuLight system with over 40,000 nodes [77], is likely to continue into the far future to tackle the challenges associated with the ending of Moore's law. The interconnection network, used to tightly couple these HPC systems together, faces increased demands for ultra-low latency from traditional scientific applications, as well as new demands for high throughput of very large messages from emerging deep learning workloads. While the commonly deployed Clos or Fat-Tree network topologies could provide the throughput, their cost structure is prohibitive and the observable latency will suffer from the increasing number of intermediate switches when scaling up the tree levels. Furthermore, as signaling rates are headed beyond 50 Gbps (greater than 200 Gbps per 4X port), "electrically-optimized", low-diameter topologies like the various "flies" — e.g., Dragonfly [41], Dragonfly+ [74], or Slimfly [9] — may no longer be relevant.



(a) 4-ary 2-tree with 16 compute nodes (b) 2-D 4x4 HyperX with 32 compute nodes (c) 28-node rack with dual-plane network

Figure 2: Two full-bisection bandwidth networks (Fig. 2a: indirect Fat-Tree topology; Fig. 2b: direct HyperX topology) and Fig. 2c shows one of 24 racks of our 672-node supercomputer with two edge switches connecting to the 3-level Fat-Tree and four switches for the 12x8 HyperX network (brown = rack-internal passive copper cables; gaps used for cable management)

In a future driven by co-packaged optics and fiber shuffles, the HyperX topology [3] could be the alternative to provide high-throughput and low network dimensionality, with the most competitive cost-structure. One drawback of these direct and/or low-diameter topologies is, however, the potential bottleneck between two adjacent switches, usually bypassed by non-minimal, adaptive routing [2]. The adverse effect of static, minimal routing for a HyperX is visualized in Figure 1. The average observable intra-rack bandwidth for bisecting communication patterns is with 2.26 GiB/s close to the maximum on a Fat-Tree (left), whereas this average drops to mere 0.84 GiB/s on a HyperX (middle). The cause: up to seven traffic streams may share a single cable in the default HyperX network configuration with minimal routing. Our non-minimal, static PARX routing, developed for the HyperX, alleviates this issue, as shown in the right-most heatmap of Figure 1, boosting the average bandwidth by 66% to 1.39 GiB/s per node pair.

In this study, we empirically quantify and analyze the viability of the HyperX topology — so far only theoretically investigated — by rewiring a large-scale, multi-petaflop, and recently decommissioned supercomputer. The 1st network plane of the original system is kept as Fat-Tree topology, while the 2nd plane is modified to facilitate our fair comparison. We stress both network topologies with numerous synthetic network benchmarks, as well as a broad set of scientific (proxy-)applications, e.g., sampled from the Exascale Computing Project (ECP) proxy applications [19] and other procurement benchmarks used by the HPC community. To mitigate the lack of adaptive routing, and resulting bottlenecks, in our dated generation of InfiniBand (QDR type), we develop and test different strategies, i.e., rank placements and our novel communication-/topology-aware routing algorithm. In short, the contributions of our paper are:

(1) We perform a fair, in-depth comparison between HyperX and Fat-Tree using more than a dozen different HPC workloads,
(2) We develop a novel topology- and communication-aware routing algorithm for HyperX, which mitigates bottlenecks induced when routing along shortest paths, and
(3) We demonstrate that even a statically routed HyperX network can rival or outperform the more costly Fat-Tree topology for some realistic HPC workloads.

2 SYSTEM ARCHITECTURE & TOPOLOGIES

Here, we briefly introduce the state-of-the-art Fat-Tree, used by many HPC systems, and the recently proposed HyperX alternative. Furthermore, we show how we re-wired a decommissioned system to perform a 1-to-1 comparison between these two topologies.

2.1 k-ary n-tree Topology

The k-ary n-tree topology [66], also known as Folded Clos or Fat-Tree, has become the predominant topology for large scale computing systems. This topology derives from multistage Clos networks [12], and has risen to dominance mainly because of its ability to support efficient deterministic routing wherein packets follow the same path every time from source to destination. In order to support full throughput for uniform random traffic, a Folded Clos must be provisioned with 100% bisection bandwidth. While this yields a system with predictable behavior, its cost is high and often prohibitive due to the indirect nature of this topology and increase in required levels n for larger supercomputers. Figure 2a shows a small 4-ary 2-tree. To reduce cost, large systems can be deployed with less bisection bandwidth by oversubscribing the lowest level of the tree [47]. For example, a 2-to-1 oversubscription cuts the network cost by more than 50% however reduces the uniform random throughput to 50%. The benefit to the Folded Clos is that all admissible (non-incast) traffic patterns theoretically yield the same throughput [66], assuming the routing is congestion-free [30].

2.2 HyperX Topology

Ahn et al. introduced the HyperX topology [3] as a generalization to all flat integer lattice networks where dimensions are fully connected, e.g., HyperCube, Flattened Butterfly, see [2, Sec. 3.3] for details. Since the HyperX is a superset of these topologies, the remainder of this paper will use the term HyperX, as the methodologies presented herein apply to all HyperX configurations. Figure 2b depicts a full-bisection bandwidth, 2-dimensional HyperX. The HyperX methodology is designed as a low-diameter, direct network to fit with high-radix routers. One of the primary benefits of HyperX is that it can fit to any physical packaging scheme as each dimension can be individually augmented to fit within a physical packaging domain, e.g., a chassis, a rack, a row of racks, etc. A HyperX network is designed for uniform random traffic being the

average case. A HyperX network designed with only 50% bisection bandwidth can still provide 100% throughput for uniform random, assuming appropriate message routing, and hence drastically reduce overall network costs. This advantage in cost structure was shown by Ahn et al., as well as in subsequent studies [6, 40, 56]. However, unlike the Folded Clos, even though 100% throughput is achievable with uniform random traffic on a HyperX, the worst case traffic will only achieve 50% throughput [13].

2.3 HPC System Layout and Modifications

Currently, no large-scale deployment of the HyperX topology exists, hence we have to build one ourselves. Fortunately, the Tokyo Institute of Technology decommissioned their TSUBAME2 supercomputer recently, and allowed us to modify its topology to construct a HyperX prototype. The system's original configuration, before being shut down, consisted of over 1,500 compute and auxiliary nodes — totalling a theoretical peak performance of ≈6 Pflop/s — interconnected by two QDR InfiniBand-based (IB) full-bisection bandwidth Fat-Trees [24]. Each compute node is equipped with two hexa-core Intel CPUs (Westmere EP generation) and at least 4 GiB RAM per core, but primarily gains the compute capability from GPU acceleration[1].

Both IB networks are constructed from 36-port Voltaire 4036 edge switches (two per rack per network plane), a total of 12 Voltaire Grid Director 4700 switches, and thousands of QDR active optical cables (AOC) between the edge and director switches. The original full-bisectional 18-ary 3-trees were undersubscribed, i.e., each edge switch hosts only 15 compute nodes (instead of 18). Tearing down one of the two network planes and re-wire it as HyperX topology, as indicated in Figure 2c, theoretically allows for an accurate and fair comparison of Fat-Tree vs. HyperX, since both physical IB network cards are attached to CPU0. Unfortunately, our InfiniBand hardware will be a weak point of our HyperX topology.

The HyperX was intended to be used and initially proposed together with the Dimensionally-Adaptive, Load-balanced routing algorithm (DAL). Our dated QDR-based InfiniBand hardware only supports flow-oblivious, static routing — usually along shortest paths — and entirely lacks adaptive routing (AR) capabilities, see [34, Sec. 18.2.4.3]. Hence, to alleviate the lack of AR and improve throughput especially for adversarial traffic, see previous Sections 1 and 2.2, we develop two mitigation strategies in Section 3.

In a multi-months effort, we constructed the largest possible HPC system — given our hardware constraints — with HyperX network resulting in a 12x8 2D topology with 7 nodes per switch, and slightly over half-bisection bandwidth, i.e., 57.1% to be precise. The 672 compute nodes and 96 IB edge switches, composing our HyperX network, are distributed over 24 compute racks (plus one auxiliary rack), giving our new system theoretically a computational peak performance of 2.7 Pflop/s (double-precision).

Initially, we planed a larger system, but various challenges impeded it: (1) extracting >900 AOCs from under a raised floor resulted in 58 broken or degraded cables; (2) the aged compute nodes suffer increased failures rates, limiting available spares (we replaced 107 nodes prior/during benchmarking); and (3) retrofitting an existing rack/node layout limits design choices. Nevertheless, we rebuilt one entire server room (of two), hosting all 24 racks, while keeping the original Fat-Tree network intact. Spare AOCs, after wiring the HyperX, are used to replace degraded cables in both topologies[2]. Unfortunately, the number of disabled cables in both networks still exceeds available spares. Consequently, 15 (out of 684) AOCs are absent from a full 12x8 HyperX, while our 204-switch Fat-Tree is missing 197 AOCs (or links inside the director switches) from a total of 2662. However, the Fat-Tree's undersubscription should limit the overall performance degradation. Hence, we end up with two slightly imperfect networks for the comparison in Section 5.

3 BOTTLENECK MITIGATION FOR HYPERX

Similar to other low-diameter topologies, such as Dragonfly or Slimfly, the HyperX relies on AR to avoid oversubscribing the shortest path. Without it, see Figure 1, the throughput is severely limited. Hence, we develop two mitigation strategies for our QDR IB technologies which only allows for static and flow-oblivious routing.

3.1 Application-to-Compute Node Mapping

The most obvious solution to the bottleneck problem, induced by static routing, is to spread out the allocation of nodes for an application, such that node-adjacent switches are either not directly connected, or only a subset of nodes of each switch is used. Ideally, one would perform a topology-/routing-aware rank mapping, see for example [1, 33]. However, these strategies are impractical in production environments, due to limited availability of idle resources. Since we do not recommend to deploy HyperX without adaptive routing, we only use a simple random assignment of ranks to nodes in this study to test if it mitigates the bottleneck. The disadvantage of this approach is an increased latency for small messages.

3.2 Non-Shortest Path Routing for HyperX

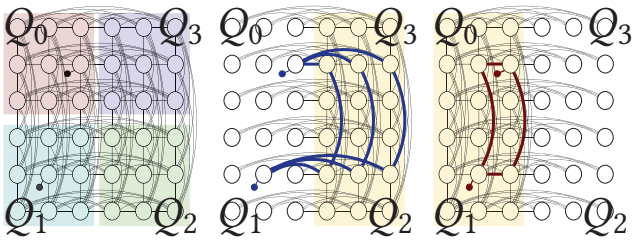
An alternative to the node mapping is altering the traffic flow to utilize more bandwidth of the network by using non-minimal paths, similar to how AR for Dragonfly topologies operates [20, 41]. However, statically routing along non-shortest paths is complicated in IB, due to the destination-based forwarding scheme employed by IB switches [34, Sec. 3.4.3]. Assume, for example, a triangle of switches A, B, and C with one node per switch, then theoretically A's node can send traffic to C's via B, but at the same time B's node cannot send traffic to C's via A, because packets would get stuck in an infinite forwarding loop between A and B. Hence, we require a novel routing algorithm for HyperX and communication scheme for applications which satisfies the following four criteria:

(1) Small messages are routed along shortest paths;
(2) Large messages utilize non-shortest paths;
(3) For all node pairs the choice between (1) and (2) exists; and
(4) The routing is loop-free, fault-tolerant and deadlock-free.

While the first three criteria obviously aim for low latency and increased throughput, criterion (4) is required due to our imperfect HyperX deployment, see Section 2. The deadlock-freedom demand became essential after initial tests with OpenSM's SSSP routing [63]. The next Sections 3.2.1–3.2.4 detail how we accomplish our goal.

[1] Facility power constraints, for running old and new HPC system simultaneously, prohibited us from using the GPUs, which should not impede our network benchmarks.

[2] We filtered out cables by creating fabric traffic and investigating its port/link error counters, e.g., one filter criterion was >10,000 symbol errors in a short time period.



(a) Quadrants (b) Forced Detours (c) Minimum Paths

Figure 3: Illustrate quadrant-based partitioning of a 2D HyperX to achieve minimal & non-minimal routing via LMC-based multi-pathing; See Sec. 3.2.1 for more details

Table 1: Valid choices of virtual destination LID_x depending on message size (s=source; d=destination; | stands for *or*); Sec. 3.2.4 for how we distinguish between small and large

(a) x for small messages

s \ d	Q0	Q1	Q2	Q3
Q0	**1 | 3**	**1**	**0 | 2**	**3**
Q1	**1**	**1 | 2**	**2**	**0 | 3**
Q2	**1 | 3**	**2**	**0 | 2**	**0**
Q3	**3**	**1 | 2**	**0**	**0 | 3**

(b) x for large messages

s \ d	Q0	Q1	Q2	Q3
Q0	**0 | 2**	**0**	**0 | 2**	**2**
Q1	**0**	**0 | 3**	**3**	**0 | 3**
Q2	**1 | 3**	**3**	**1 | 3**	**1**
Q3	**2**	**1 | 2**	**1**	**1 | 2**

3.2.1 Concurrently Forwarding along Minimal & Non-Minimal Paths. As the previous thought experiment examplifies, it can be impossible — with out-of-the-box InfiniBand routing mechanisms — to accomplish criteria (1)–(3) for the general case, i.e., larger topologies comprised of multiple switches. However, IB enables up to 128 virtual destinations, called local identifier (LID), per physical port of host channel adapters (HCA), see [34, Sec. 4.1.2]. The LID mask control (LMC) defines how many (up to 127) additional LIDs are assigned besides the base LID_0. The subnet manager, usually OpenSM, then calculates the forwarding tables as if each (virtual) LID would be a physical endpoint in the fabric [34, Sec. 3.5.10].

We utilize this multi-destination feature of IB to construct a topology-aware, static routing for our rewired supercomputer with 2D HyperX topology. Our novel approach is generalizable to higher dimensions, however due to the prototypic nature of it[3], we limit ourselves to only 2D HyperX topologies with even dimensions.

Assuming, we virtually divide the 2D HyperX into four quadrants Q0–Q3, see Figure 3a, and each HCA/port is configured with four destination LIDs (by setting LMC = 2 and indexed via $LID_0, \ldots, LID_3$), then the calculated paths towards them may be partially or fully disjoint[4]. Unfortunately, available static routing for IB will only calculate routes along the minimal paths in the network, unless it serves the purpose of deadlock-avoidance, e.g., as performed by Up*/Down* [72] or Nue routing [16]. Hence, to enforce the traffic forwarding along non-minimal paths, a routing algorithm must be topology-aware and/or temporarily disregard links which otherwise contribute to a minimal path length.

Our approach for this challenge is to virtually remove some adverse links during the path calculation. Let's illustrate this approach through an example before generalizing it for the entire HyperX: Assuming, the routing iteratively processes the destination LIDs and the "currently" processed LID_0 belongs to quadrant Q1, see Figure 3b. Now, if we virtually remove all network links within the left half of the HyperX, then it will cause all "shortest" paths originating from Q0 or Q1 to traverse through Q2 and/or Q3. The benefit is an increase of non-overlapping paths, i.e., from at most two to $\frac{D_1}{2}$, if the 1st dimension of the HyperX is larger than 4. In this example, we gain three additional paths. Large messages should utilize the increased link bandwidth to avoid creating bottlenecks on direct links in the left half. In contrast, if another virtual LID_1 is attached to the same switch in Q1 and the link removal is applied to the right half, then all paths towards LID_1 will be minimal, exemplified in Figure 3c. However, paths ending in Q2/Q3 may detour.

To generalize this example, we define the following four rules when processing a given destination LID in the routing engine:

(R1) given LID ≡ LID_0 ⇒ remove all links in left half
(R2) given LID ≡ LID_1 ⇒ remove all links in right half
(R3) given LID ≡ LID_2 ⇒ remove all links in top half
(R4) given LID ≡ LID_3 ⇒ remove all links in bottom half

and any communication traffic injected into the network, which is addressed to arrive at node n, should select the correct virtual destination LID_x^n based on the message size and based on the numbers for x listed in Table 1. One can easily deduct from these two Tables 1a and 1b, and Figure 3, that this routing approach achieves criteria (1)–(3) for a 2D HyperX topology, such as ours[5].

3.2.2 Optimization for Communication Demands of Applications. Another advantage of AR over static routing is the ability to dynamically balance traffic flows. For the latter, a mismatch between the calculated paths and injected flows can cause congestion [30]. Tuning the static routes towards the actual communication demands of one or more applications may be beneficial — assuming a relatively sparse and reoccurring communication pattern — for emulating AR. A similar strategy, called scheduling-aware routing (SAR) [14], adds application-to-node mappings into the routing engine. Hence, we modify SAR [14] in Section 3.2.3 to ingest communication profiles.

Acquiring these communication profiles for point-to-point MPI messages [51] is trivial with tools such as Vampir(Trace) [43] or TAU [73], but for MPI's collective operations these tools only provide high-level information. Actual point-to-point messages which compose the collectives, and therefore the real node-to-node traffic demand, remain unrecorded. Hence, we rely on a low-level IB profiler [10], to record/store the profiles for each combination of benchmark, input, and number of MPI ranks[6]. Our routing engine can then perform fine-grain path balancing optimizations.

3.2.3 Pattern-Aware Routing for HyperX Topologies (a.k.a. PARX). We combine the above outlined technical aspects into one novel

[3] Future HyperX deployments use AR, making our static routing prototype obsolete.
[4] Statement depends on applied routing and omits switch-to-HCA sections of the path.
[5] We accomplish the identification of quadrants in our implementation by predefining the LID-to-{port|switch} assignment through OpenSM's *guid2lid* mapping file.
[6] This profile is immune to changed in MPI rank placement, topology, and IB routing.

Algorithm 1: Pattern-Aware Routing for 2D HyperX (PARX)

Input: Network graph $I = G(N, C)$ with nodes N and links C, Communication demand file with one line per source node $D := [(\text{<destination>}, \text{<send demand>}), \ldots]$
Result: Communication-aware, minimal & non-minimal, deadlock-free routing with valid paths P_{n_x,n_y} for all $n_x, n_y \in N$

```
/* Process of the communication demands                                  */
foreach node n ∈ N do
    n.demands ← empty list []
    foreach pair (nodeName, send demand) ∈ D[n] do
        if nodeName ≡ m ∈ N then n.demands.append[(m, send demand)]

/* Optimize routing for compute nodes listed in D, assuming 2^LMC = 4    */
foreach node n_d ∈ N with (n_d, ·) ∈ D do
    foreach i ∈ {0, ..., 3} do
        Create temporary graph I* = G(N, C*) by remove links from C according to
          LID_i of n_d and according to the rules (R1)–(R4) listed in Sec. 3.2.1
        Calculate a path P_{n_x,n_d} for every pair (n_x, n_d), with n_x ∈ N in I* by using
          the modified Dijkstra algorithm of DFSSSP routing (details in [17])
        /* Update edge/link weights in graph I before the next round     */
        foreach node n_x ∈ N do
            if (n_d, w) ∈ n_x.demands then
                Increase edge weight by +w for each link in path P_{n_x,n_d}

/* Calculate routing for all other nodes which are not listed in D       */
foreach node n_d ∈ N ∧ n_d not processed before do
    Calculate paths P_{·,n_d} for all n_x ∈ N and i = 0, ..., 3 as shown above, however
      Only update edge weights for all links used by P(·, n_d) with a +1 per path

/* Create deadlock-free routing configuration                            */
foreach path P_{n_x,n_y} calculated above (incl. all virtual LIDs) do
    Assign P_{n_x,n_y} to one virtual layer without creating a cycle in the corresponding
      channel dependency graph (see [17, 75] for details on VL-based deadlock-avoidance)
```

routing algorithm which is tailored for statically-routed, InfiniBand-based 2D HyperX topologies. PARX increases path diversity by simultaneously providing minimal and non-minimal paths and allows for communication demand-based re-routing of the fabric, all while being fault-tolerant[7] and deadlock-free[8].

We implement PARX in IB's subnet manager (OpenSM) by taking the deadlock-free SSSP routing (DFSSSP) [17, 63] as basis, and modify it by adding an altered version of the SAR extensions [14] — to ingest communication profiles instead of rank-to-node mappings. Furthermore, we changed DFSSSP's path calculation to temporarily ignore/remove links from the network to abide by the rules (R1)–(R4), and to consider the data from the communication profiles for path balancing and tuning purposes. The pseudo-code of the PARX routing engine for our HyperX is shown in Algorithm 1.

The communication profiles contain the absolute number of bytes transferred between every pair of MPI ranks during the program execution. We normalize these (potentially large) numbers to the integer range of $D_n := [0, \ldots, 255]$, where 0 stands for absolutely no bytes transferred between two ranks. A 1 indicates a relatively low amount of bytes and 255 represents the highest traffic demand between two MPI ranks. These normalized traffic demands are used by PARX to balance the routes assigned to links such that the number of overlapping paths, carrying high traffic demands, are minimized. The base algorithm, DFSSSP, alternates between calculating all paths towards one destination LID_x and performing edge update of the weighted graph representing the network topology. Per calculated path, DFSSSP adds +1 to each link along the path [17]. Hence, DFSSSP's edge update results in global path balancing, oblivious to the workload on the HPC system. In contrast, our PARX algorithm updates the edges by adding $w \in D_n$, resulting in per-application(s) optimized paths, see the inner-most loop of the triple nested loop in Algorithm 1. This approach reduces the dark fiber [14], and high-traffic paths are separated as much as possible to reduce congestion observed by the applications.

3.2.4 Modifications to the Message Passing Interface (MPI) Library. The criteria (1) and (2) listed in Section 3.2, and routing approach with PARX, require a categorisation of messages injected into the network, as well as the assignment of appropriate (virtual) destination LIDs for these messages. None of the existing MPI libraries is capable of performing this task. While, Open MPI [23] supports IB's multi-LID addressing, the default configuration — with the *ob1* point-to-point messaging layer (PML) — uses multiple LIDs only for fail-over in case of connection issues on the primary path.

The alternative PML, called *bfo*, offers concurrent multi-pathing for IB by setting up as many connections between two HCAs as these are (virtual) LIDs assigned to them, i.e., LID_0 of port s can communicate with LID_0 of port d, LID_1^s with LID_1^d, etc. The bfo PML iterates through the 2^{LMC} LIDs in a round-robin fashion. After transferring a message (or message segment for larger messages) to LID_x the layer increments x or resets to 0. Hence, we can easily modify this bfo point-to-point messaging layer to set x based on the HyperX quadrants[9] and rules provided in Table 1. Whenever Table 1 lists two alternatives, we randomly select one.

In addition to the quadrant identification for a give injected message ($s \rightarrow d$), we need to distinguish message sizes for the selection of x for the virtual LID_x^d. We performed an initial test with Intel's Multi-PingPong MPI benchmark [35] and mpiGraph, to evaluate at which node-count per switch and which message size we observe latency degradation due to congestion on the single link between the two involved HyperX switches. Consequently, we define the threshold to be 512 bytes for all PARX-based evaluations[10].

4 METHODOLOGY

While we could deduce the suitability of the HyperX topology for HPC applications from simple MPI benchmarks, actually testing a broad spectrum of real scientific/HPC workloads, as listed below, will refine our understanding of the novel topology. The following inputs are tuned using a smaller node count such that each test should theoretically take ≈1–5 min, regardless of scale.

4.1 Pure MPI/Network Benchmarks

We evaluate raw latency/throughput performance for small messages, as found in HPC codes [42], and large communication loads, as required for deep learning applications, with three benchmarks:

- *Intel MPI Benchmarks (IMB)* perform network latency/throughput measurements of point-to-point and collective MPI operations of varying message sizes [35]. We focus on IMB's single-mode MPI-1 collectives (non-v version), meaning Barrier, Bcast, . . ., Alltoall.

[7] Fault-tolerance is limited, because temporarily removing links, see Sec. 3.2.1, can result in unreachable LIDs if the adjacent switch has no remaining links.
[8] For all of our evaluations, see Sec. 5, PARX requires between 5 and 8 virtual lanes (VLs), depending on ingested communication profile, which is within limit of 8 available VLs for our IB hardware. PARX may exceed a VL hardware limit for larger HPC systems.

[9] As done in PARX routing, we identify the quadrants by an appropriately predefined LID-to-port assignment and determine quadrant q via equation $q := \lfloor \frac{\text{LID}}{1000} \rfloor$.
[10] This threshold depends on interconnect technology and #{nodes} attached to each HyperX switch. Determining an optimal threshold is beyond the scope of this paper.

• *Netgauge's eBB* evaluates the effective bisection bandwidth [29] for a given topology, as induced by the selected routing. We execute 1,000 random bisections with 1 MiB message size per sample.

• *Baidu's DeepBench Allreduce (AllR)* implements a ring-based allreduce and evaluates the latency for a variety of messages sizes (0–2 GiB) [7]. We evaluate the CPU-only version of the code.

4.2 Scientific Application Benchmarks

Besides pure MPI benchmarks, we cover a broad set of scientific and HPC domains by selecting Exascale Computing Project (ECP) [19] procurement benchmarks, workloads from RIKEN R-CCS' Fiber Suits [68], and two codes — known to be highly communication-intensive — from the Trinity [59] and CORAL [45] set:

• *Algebraic multi-grid (AMG)* solver of the *hypre* library is a parallel solver for unstructured grids [64] arising from fluid dynamics problems. We choose *problem 1* with a 256^3 cube per process for our tests, which applies a 27-point stencil on a 3-D linear system.

• *Co-designed Molecular Dynamics (CoMD)* serves as the reference implementation for ExMatEx [54] to facilitate co-design for (and evaluation of) classical molecular dynamics algorithms. We are using the included weak-scaling example to calculate the inter-atomic potential for 64^3 atoms per process.

• *MiniFE (MiFE)* is a reference code of an implicit finite elements solver [27] for scientific methods resulting in unstructured 3-dimensional grids. For our study, we follow the recommended $n_x = n_y = n_z = \sqrt[3]{n_{x_b} * n_{y_b} * n_{z_b} * \#processes}$ weak-scaling formula with $n_{\{x|y|z\}_b} = 100$ to define the grid's input dimensions.

• *SWFFT (FFT)* represents the compute kernel of the HACC cosmology application [26] for N-body simulations. The 3-D fast Fourier transformation of SWFFT emulates one performance-critical part of HACC's Poisson solver. In our tests, we perform 16 repetitions on a 3-D grid, which is weak-scaled similar to [78, Tab. 4.1].

• *Frontflow/violet Cartesian (FFVC)* uses the finite volume method (FVM) [62] to solve the incompressible Navier-Stokes equation for thermo-fluid analysis. Here, we calculate the 3-D cavity flow in a 128^3 cuboid per process for weak-scaling.

• *many-variable Variational Monte Carlo (mVMC)* method implemented by this mini-app is used to simulate quantum lattice models for studying the physics of condensed matter [53]. We use mVMC's included weak-scaling test (*job_middle*) without modifications.

• *NTChem (NTCh)* implements a computational kernel of the software framework (NTChem) for quantum chemistry calculations of molecular electronic structures, i.e., the solver for the second-order Møller-Plesset perturbation theory [58]. We select the provided *taxol* test case for our study as strong-scaling input.

• *MIMD Lattice Computation (MILC)* is performing quantum chromodynamics (QCD) simulations using the lattice gauge theory on the Lie group SU(3) [8]. We use NERSC's Trinity MILC benchmark code and weak-scale their single node *benchmark_n8* input [60].

• *LLNL's qb@ll (Qbox)* is an improved Qbox version [25, 44] for first-principles molecular dynamics, which uses Density Functional Theory (DFT) to, for example, calculate the electronic structure of atoms. We weak-scale the computational load of qb@ll's included *gold* benchmark, assuming a single-node case of 32 gold atoms.

4.3 *x*500 Benchmarks

Lastly, we employ three HPC benchmarks[11], which the community uses to compare the supercomputers in a world-wide ranking:

• *High Performance Linpack (HPL)* is solving a dense system of linear equations $Ax = b$ to demonstrate the double-precision compute capabilities of a (HPC) system [77]. Our problem size is tuned such that matrix A occupies ≈1 GiB per process.

• *High Performance Conjugate Gradients (HPCG)* is applying a conjugate gradient solver to a system of linear equations (sparse matrix A) [18], to demonstrate the system's memory and network limits. We choose 192×192×192 as process-local problem domain.

• *Graph 500 Benchmark (GraD)* measures the data analytics performance of (super-)computers by evaluating the traversed-edges-per-second metric (TEPS) for a breadth-first search (BFS) on a large graph [57]. Our input graph occupies ≈1 GiB per process and we perform 16 BFSs with a highly optimized implementation [79].

4.4 Test Strategies, Environment, and Metrics

The benchmarks and applications, listed in Section 4.1–4.3, will be evaluated in two different settings, i.e., in isolation to show system capability, and in a more realistic multi-application environment.

4.4.1 Capability Evaluations. Our exclusive system access allows us to execute capability runs sequentially and without overlap, while keeping unoccupied nodes idle, which should give insight into idealized achievable performance on the given topology. We scale each benchmark starting from a single switch, i.e., seven nodes, or four nodes if the benchmark requires #nodes in power-of-two. From there we double the node count in each step, up to the maximum possible node count, i.e., 7, 14, . . . , 448, 672 or 4, 8, . . . , 512, respectively. Each combination of: benchmark (and scale), topology, routing, and placement (cf. Section 4.4.3), is executed ten times to capture the best performance and occurring run-to-run variability.

4.4.2 Capacity Evaluations. The multi-application execution model is more common for many supercomputers [69]. These concurrently running jobs may compete for bandwidth, or create inter-job interference which can increase message latency [37]. We select the following applications: AMG, CoMD, FFVC, Graph500, HPCG, HPL, MILC, MiniFE, mVMC, NTChem, qb@ll, and SWFFT, plus one IMB Multi-PingPong (*MuPP*) and one modified IMB Allreduce (*EmDL*) benchmark[12]. Each application gets a dedicated set of nodes (32 or 56 nodes, respectively), and all are submitted simultaneously and are configured to execute thousands of runs. We let the supercomputer perform this capacity evaluation for 3 h, using 664 of the 672 available compute nodes[13]. We evaluate the number of runs per application and compare these numbers across the different topologies, routings, and allocations.

4.4.3 Routing and Placement. Both the routing algorithm and the MPI rank placement can positively (or negatively) influence the

[11] For both HPL and HPCG, we employ the highly tuned versions shipped with Intel's Parallel Studio XE (v2018; update 3) with appropriate *PNB* parameter for our system.
[12] EmDL is a modified IMB Allreduce to mimic deep learning workloads by alternating between communication and an 0.1 s compute phase simulated via usleep.
[13] Designed as qualitative comparison between the two topologies (due to complexity of such evaluations) to look for potential weaknesses in our HyperX to aid future R&D.

communication performance for parallel applications. Hence, we evaluate five different routing and placement combinations.

We choose the commonly used *ftree* routing [85], as well as SSSP routing [31] (both part of IB's OpenSM [34]), for the Fat-Tree. The latter theoretically yields increased throughput for faulty Fat-Tree deployments [15] such as ours (cf. Section 2.3). Furthermore, we test two MPI rank placements for the Fat-Tree. The first placement is a *linear* assignment of MPI ranks, meaning rank 1 is placed on compute node n_1, ..., rank i on node n_i, and so on, which is a common resource allocation practise [71, 84]. It reduces small message latency while isolating small-scale runs into subpartitions of the network to reduce interference [52]. The *clustered* placement is more realistic and yields from the system's fragmentation over its operational period [67]. We simulate it by drawing the stride Δ from node n_i to the next node n_j from a geometric distribution with an (arbitrarily chosen) 80% probability, and hence $j := i + \Delta$.

We rely on OpenSM's DFSSSP [17] for our HyperX network due to aforementioned deadlock-issues, see Section 3.2. DFSSSP requires only 3 virtual lanes (VL) to achieve the deadlock-freedom for our HyperX, which is well within the hardware limit of 8 VLs. Additionally, we test our novel PARX routing for bottleneck mitigation, see Section 3.2.3, as well as the *random* rank placement introduced in Section 3.1. Furthermore, we test *linear* and *clustered* placements. Our stored communication profiles are MPI rank-based and placement oblivious, therefore we require an interface — similar to SAR's [14] — between the job-submission and OpenSM. This interface combines the profile(s) and selected node allocation for one (or more) application into a node/LID-based demand data file, which PARX uses to re-route the fabric prior to the job start.

In summary, in Section 5 we are evaluating the following five combinations of topology, routing, and resource allocation scheme:

(1) Fat-Tree with ftree routing and linear placement;
(2) Fat-Tree with SSSP routing and clustered placement;
(3) HyperX with DFSSSP routing and linear placement;
(4) HyperX with DFSSSP routing and random placement; and
(5) HyperX with PARX routing and clustered placement,

and collect performance data from all runs, as indicated below.

4.4.4 Evaluated Performance Metrics. We extract the performance data directly from the output of the pure network benchmarks (cf. Section 4.1), i.e., observable communication latency and message throughput for different MPI operations and messages sizes. Details about the collected metrics is provided in Table 2, which also summarizes how we scale up each (application-)benchmark. For reference purposes, we include the executed MPI communication functions used by each benchmark. The same methodology is used for the *x*500 benchmarks (cf. Section 4.3), which directly report either floating-point operations per seconds or median traversed graph edges per second. In contrast, for the nine HPC workloads listed in Section 4.2, we uniformly report the runtime of the main computational solver/kernel[14]. Focusing on only the solver phase is required, because (for most proxy-apps) the pre- and/or post-processing phase is disproportionally long which skews the performance expectations for the real applications.

[14]We injected timing instructions into the original code, whenever the proxy-app did not provide accurate timings of the solver phase or reported alternative metrics.

Table 2: List of applications/benchmarks; Overview of used MPI functions; Collected metrics from each BM; Deployed scaling method (*: instances where we scaled down the input for larger #nodes to reduce runtime; further details in Sec. 5)

MPI	Used MPI point-to-point & collective functions	Scaling	Metric
IMB	(All)Reduce Alltoall Barrier Bcast Gather Scatter	weak	Latency t_{min} [μs]
eBB	Isend Irecv Barrier Gather Scatter	strong	Throughput [MiB/s]
AllR	Send Irecv Sendrecv Allgather	weak	Latency t_{avg} [s]
Apps	**Used MPI point-to-point & collective functions**	**Scaling**	**Metric**
AMG	(I)Send (I)Recv Allgather(v) Allreduce Bcast etc.	weak	Kernel runtime [s]
CoMD	Sendrecv Allreduce Barrier Bcast	weak	Kernel runtime [s]
MiFE	Send Irecv Allgather Allreduce Bcast	weak	Kernel runtime [s]
FFT	(I)Send (I)Recv Allreduce Barrier	weak	Kernel runtime [s]
FFVC	Isend Irecv (All)Reduce Gather	weak*	Kernel runtime [s]
mVMC	(I)Send Sendrecv Recv (All)Reduce Bcast Scatter	weak	Kernel runtime [s]
NTCh	Isend Irecv Allreduce Barrier Bcast	strong	Kernel runtime [s]
MILC	Isend Irecv Allreduce Barrier Bcast	weak	Kernel runtime [s]
Qbox	(I|R)Send (I)Recv (All)Reduce Alltoallv Bcast etc.	weak*	Kernel runtime [s]
***x*500**	**Used MPI point-to-point & collective functions**	**Scaling**	**Metric**
HPL	Send (I)Recv	weak*	Floating-point Op/s
HPCG	Send Irecv Allreduce Alltoall(v) Barrier Bcast	weak	Floating-point Op/s
GraD	Isend Irecv Allgather (All)Reduce(_scatter) etc.	weak	Traversed edges/s

4.4.5 Execution Environment. Our HPC system uses the CentOS 7.4 operating system for the compute nodes, configured for diskless operation, and the OpenFabrics Enterprise Distribution (OFED) stack (version 4.8) for the IB networks, with one exception: a renewed OpenSM (v3.3.21). OpenMPI 1.10.7 serves as communication library. We refrain from adding other (usually needed) HPC software components, such as batch scheduler or parallel file system[15].

All applications and micro-benchmarks are compiled with the OS-provided Gnu compilers[16]. We refrain from modifying the default, application-provided compiler options, and only additionally optimize for our CPUs by specifying the `-march=native` flag.

The general execution model for our benchmarks is MPI+OpenMP with one MPI rank per compute node and one OpenMP thread per physical CPU core. The OpenMP threads are pinned and MPI ranks are mapped sequentially onto the nodes provided by a sorted `hostfile`. While there are certainly more optimal (in terms of computational performance) configurations for some of the applications — with more ranks per node or thread-to-core under- or oversubscription — we focus in our study on the 1-to-1 network comparison for which our rank/thread model should be sufficient. Each benchmark invocation is allowed a 15 min walltime before being forcefully terminated, to prevent excessive overruns.

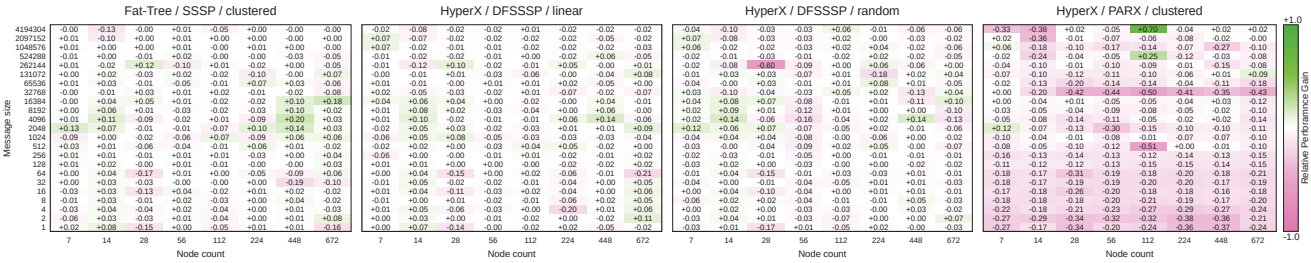
5 EVALUATION

The following measurements are performed according to the stipulated terms of Section 4.4. Therefore, the only difference between benchmarks — besides replaced nodes — is the fabric and placement.

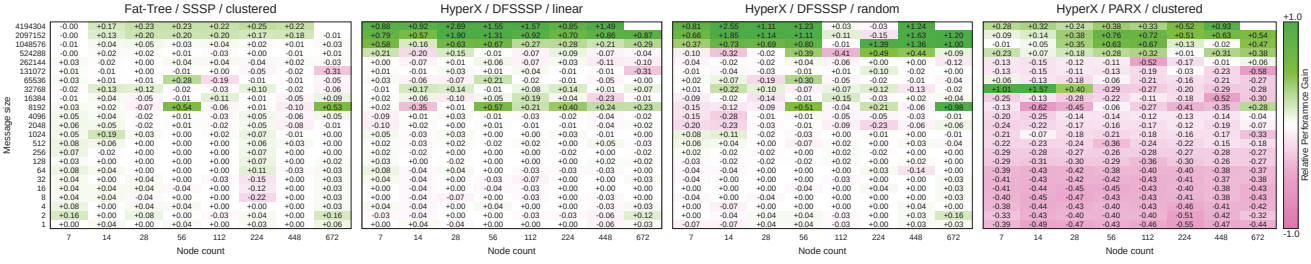
5.1 Network Benchmarks

For IMB's MPI collective benchmarks, shown in Figure 5, we extract the absolute best observed communication latency (i.e., $t_{min}[in\ \mu s]$) for each message size from our 10 runs for the five topology, routing,

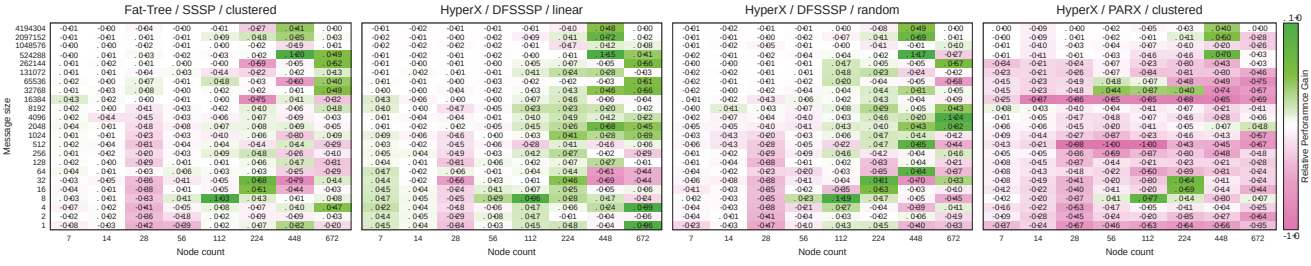
[15]The network filesystem (NFS) is able to handle the miniscule I/O of the proxy-apps, and for simplicity/repeatability, we rely on `hostfiles` and manual execution of jobs. While our software stack may appear dated, it roughly matches the environment when the system was in production. Please, refer to the AD/AE appendix for further details.
[16]See footnote 11 in Secion 4.3 for the two exceptions in the compiler selection.



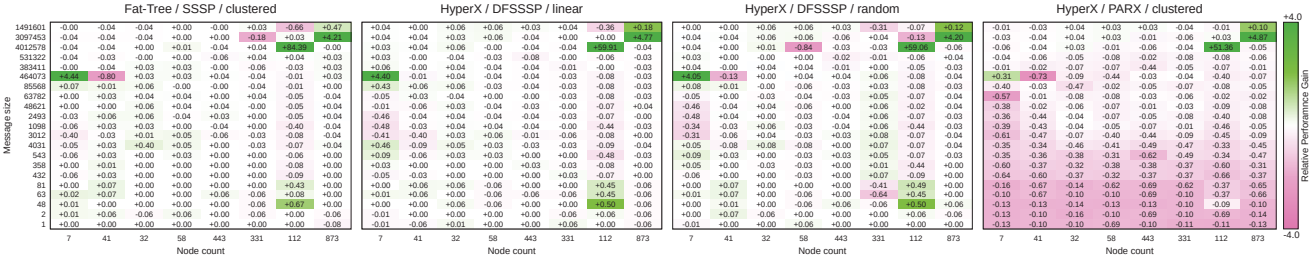
(a) Bcast



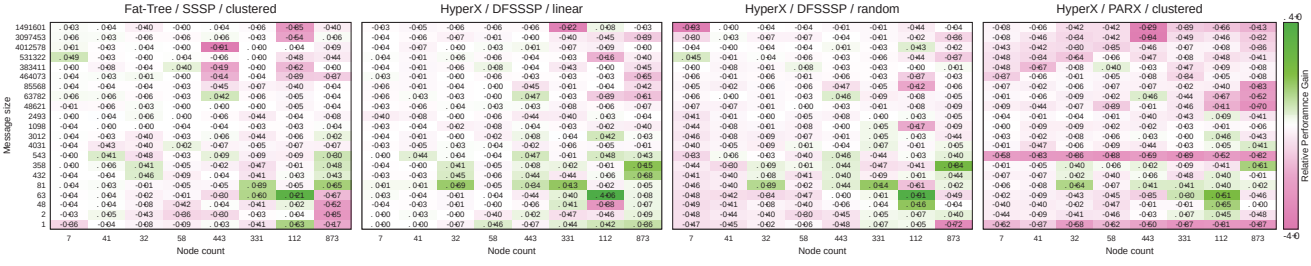
(b) Gather



(c) Scatter

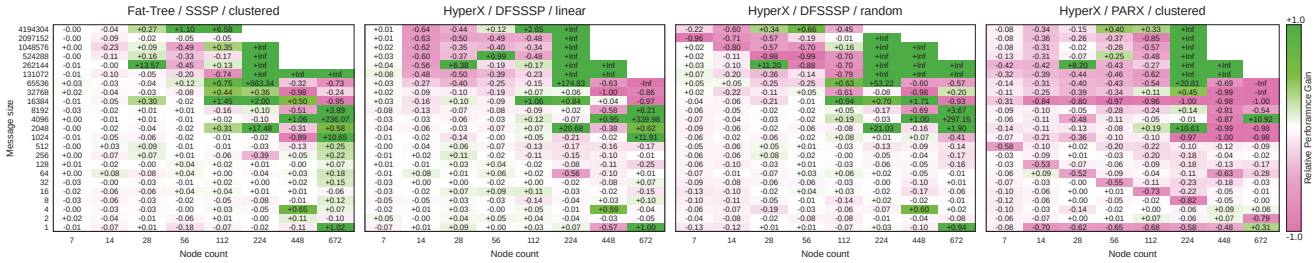


(d) Reduce



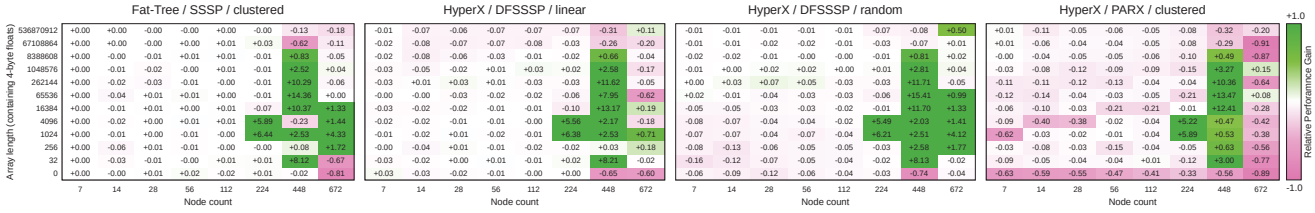
(e) Allreduce

Figure 4: Relative performance gain compared to Fat-Tree / ftree / linear combination (cf. Sec. 4.4.3) for various MPI collectives

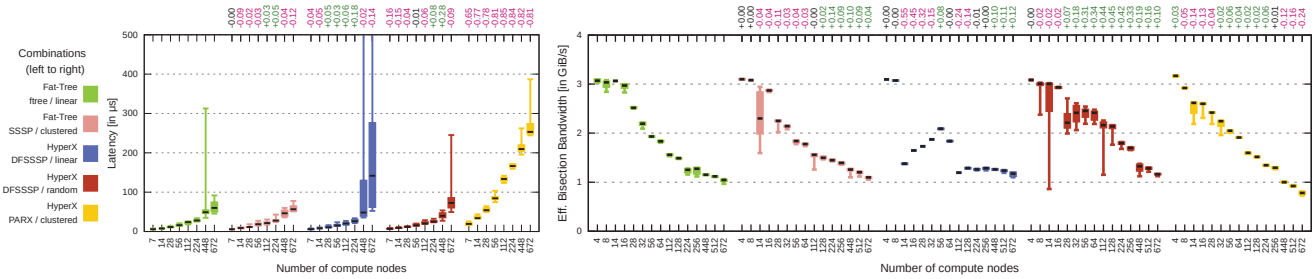


(f) Alltoall

Figure 4: Relative performance gain compared to Fat-Tree / ftree / linear combination (cf. Sec. 4.4.3) for various MPI collectives



(a) Baidu's Allreduce (relative gain over Fat-Tree / ftree / linear)



(b) IMB's Barrier **(c) Netgauge's eBB**

Figure 5: Rest of network benchmarks; Whisker plots for 10 runs show in Fig. 5b and 5c (numbers above each Figure indicated relative performance gain over Fat-Tree / ftree / linear for the best run of both combinations); See Sec. 5.1 for detailed analysis

and placement configurations. All absolute best values are then compared relatively to our baseline configuration, which is Fat-Tree with ftree routing and linear placement. The relative gain (or loss) [28] is highlighted graphically and numerically in each box.

Our HyperX network with either DFSSSP routing or randomized placement appears to be on par with the baseline for Broadcast and Reduce (Figure 4a, 4d), while outperforming the Fat-Tree for Gather with messages of 512 KiB or more, see Figure 4b, or for Scatter on larger node counts and almost all message sizes, see Figure 4c.

In contrast, we observe that MPI Allreduce — at least for messages below 4 MiB — is slightly biased towards our 3-level Fat-Tree. Baidu's ring-based Allreduce implementation (cf. Figure 5a) reveals a noteworthy problem with ftree routing, but not Fat-Tree itself, since SSSP mitigates the problem equally well as the HyperX.

The Alltoall performance (visualised in Figure 4f; missing boxes indicate time or memory constraints), shows the highest spikes in both directions. The 14-node case for "HyperX / DFSSSP / linear", for example, echos exactly our analysis of Figure 1. These 14 nodes are attached to one Fat-Tree switch, but attached to two HyperX switches which are interconnected by only a single QDR link.

On average, our PARX routing is the least effective options for these micro-benchmarks shown in Figure 4, especially for the lower spectrum of investigated message sizes. It should be noted, that we are switching from the relative data presentation to a direct comparison based on whisker plots — showing minimum, maximum, median, and 25th/75th percentile of our ten runs per measurement configuration — starting from Figure 5b. The relative gain over Fat-Tree is still visible in the numbers listed above each plot. Looking at Figure 5b, we see that PARX slows down the Barrier operation by 2.8x–6.9x, resulting in negative gains between -0.65 and -0.85 compared to the baseline. Specifically, the 7-node case (for which clustered and linear `hostlists` are identical, i.e., all nodes are attached to 1 HyperX switch) suggests a severe performance loss when moving from ob1 to the bfo PML (see Section 3.2.4 for details on our bfo dependency). We speculate that bfo is less tuned compared to the ob1 default, and hence, is likely the root cause of

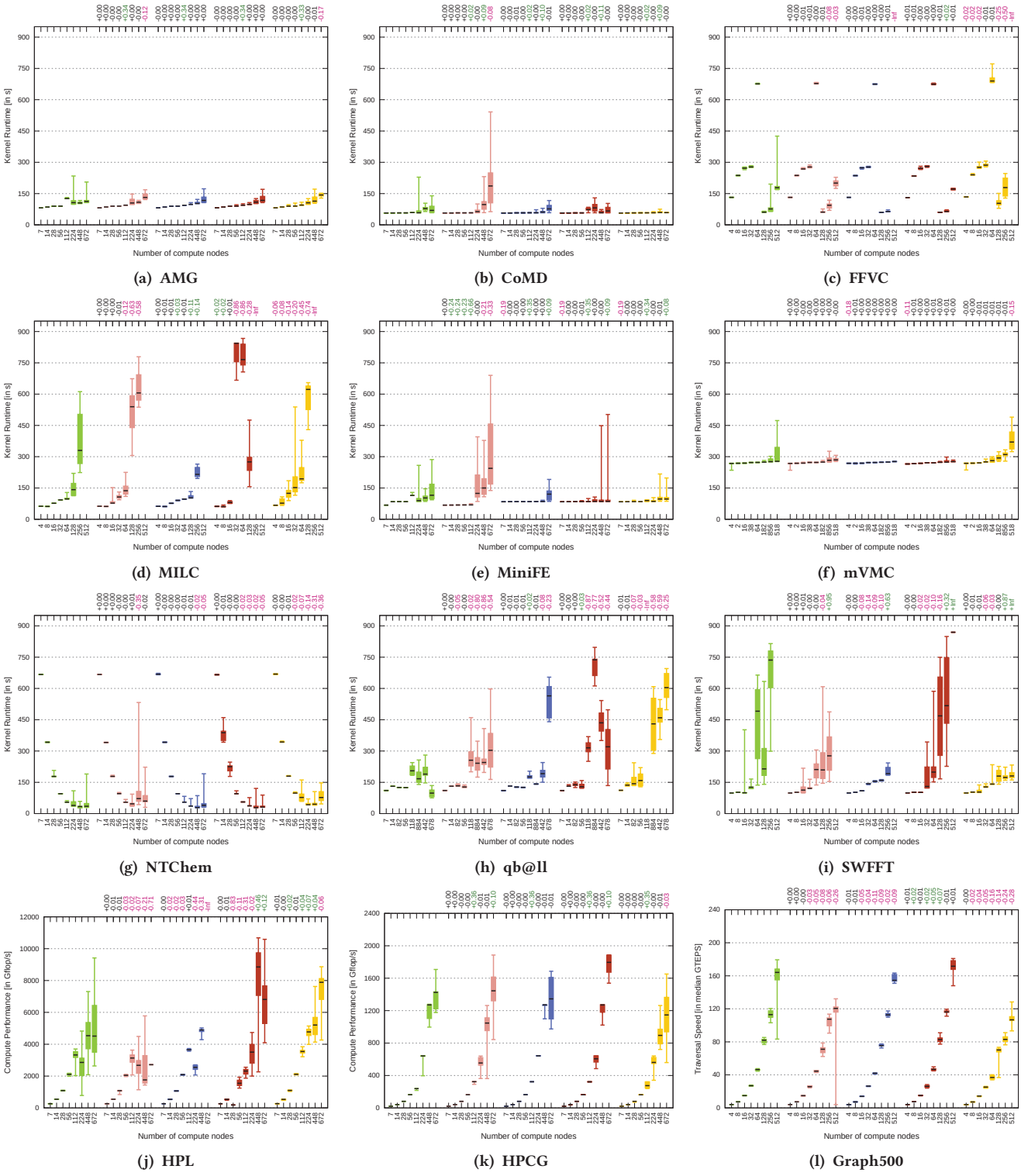


Figure 6: Proxy-applications (Fig. 6a–6i; lower is better) and *x*500 benchmarks (Fig. 6j–6l; higher is better); Whisker plots for 10 runs; Missing data points for runs exceeding 15 min time limit; Legend same as in Fig. 5b; See Sec. 5.2 for detailed analysis

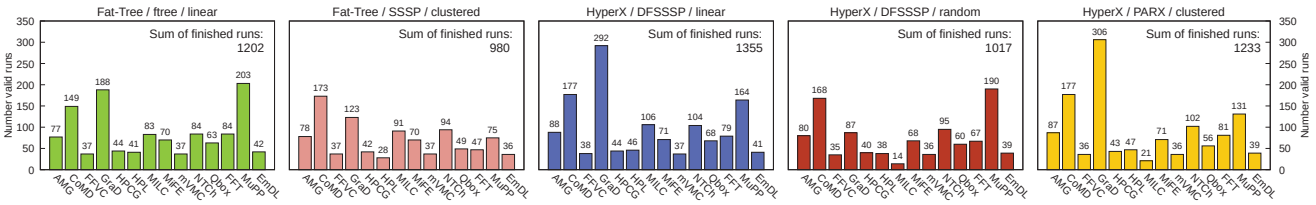


Figure 7: Capacity run for all five combinations (cf. Sec. 4.4.3) over a 3 h time period for 14 concurrently running applications (using 32 or 56 nodes) while occupying 98.8% of the supercomputer; See Sec. 4.2 for abbreviations and Sec. 5.3 for data analysis

aforementioned regression for Barrier and other collectives. Our theory is further strengthened by analyzing the MPI Gather and Scatter plots. One can see that, with increasing message sizes, the disparity to the "Fat-Tree / ftree / linear" baseline diminishes.

For 1 MiB messages of Netgauge's eBB benchmark, where software overhead diminishes, we see the advantage of the non-minimal routing by PARX compared to DFSSSP for dense node allocations on the HyperX. Especially, for the previously discussed 14-node case, we almost double (≈1.9x) the effective bisection bandwidth. Furthermore, PARX outperforms Fat-Tree / ftree (with 2%–6%) for the mid-range of the nodes counts. We expected the shown regression of PARX for full system runs of eBB, since artificially increasing the path length for large messages creates more congestion on a global scale, instead of avoiding localized bottlenecks. Whether these characteristics for pure MPI benchmarks translate into similar results for applications, is subject of the following two sections.

5.2 Real-world Workloads & *x*500 Benchmarks

Our whisker plots in Figure 6 comprise the results for the proxy-applications, where runtime of the solver/kernel is shown (lower is better in Fig. 6a–6i). In addition, Figures 6j–6l present the computational performance (higher is better) for the *x*500 benchmarks. Furthermore, all benchmarks, except NTChem, demonstrate weak-scaling behavior based on the inputs outlined in Section 4.

It should be noted, that we modify the scaling for three benchmarks, listed in Table 2. To complete runs within the given 15 min walltime limit, we reduce FFVC's cuboid to $64 \times 64 \times 64$ for runs on more than 64 nodes. The resulting runtime drop from 64 to 128 nodes is clearly visible. Similarly, we reduce qb@ll's initial input from 32 atoms to 16 gold atoms for the 672-node evaluation, and we shrink the matrix for HPL down to 0.25 GiB per process for 224 nodes and beyond. MILC on 512 nodes resists all attempts to get the runtime within the limit, so we omit the 512-node case.

The Figure 6 shows, when looking at the best of ten iterations, that our HyperX network — with either appropriate routing or placement strategy — is on par with the Fat-Tree baseline. The examples, where the HyperX mostly performs within ±1% (or notably better), include: AMG, FFVC, MILC (with "DFSSSP / linear"), MiniFE, mVMC, and NTChem / qb@ll (both for small and mid-scale runs). Moreover, we experience a reduction in run-to-run variability in some instances and when using the HyperX fabric, most conspicuous for MILC and SWFFT when using DFSSSP routing together with the linear rank placement. The latter benchmark includes another interesting and unexpected – based on previously shown results — data point: The HyperX together with PARX routing is the only option to consistently scale the SWFFT benchmark to 512 compute nodes. Here, all 10 executions finish in under 233 s.

While our PARX routing results in a clear communication latency disadvantage over the alternatives for pure network benchmarks, we see a less severe, but noticeable, impact of the less tuned bfo PML for real-world workloads. The reason is twofold: firstly, these applications only spend a fraction of the runtime in communication routines (typically an average of 20% if sampled across many proxy-applications [42]), and hence the communication performance decrease is overall less salient. Secondly, pure MPI benchmarks, including our results shown in Figure 4, tend to focus on the best case rather than the average performance, which can be influenced by system noise [32] and caching effects [55].

The results for the remaining three benchmarks, HPL, HPCG, and Graph500, induce comparable inferences. For example, the random rank placement on the HyperX for HPL improves the achievable Gflop/s performance by 46%, but our relatively small input matrices (to reduce runtime) and run-to-run variability might account for this discrepancy. Although, we also see improvements of up to 36% and 7% for HPCG and Graph500, respectively, when randomly assigning nodes and using HyperX with DFSSSP routing.

5.3 Capacity / System Throughput Evaluations

The following experiments, as outlined in Section 4.4.2, are designed as qualitative comparison between the two topologies to potentially identify weaknesses of the HyperX which can aid future research or development. For a quantitative comparison, one should use simulation-based evaluations instead, similar to Yang et al. [83], since these offer repeatability and fine-grained inspection capabilities into timings, routers, traffic, and congestion, etc.

Figure 7 shows the number of completed application runs in the given 3 h time period, and compared across our five test setups. While the random placement with DFSSSP results in the highest gain over Fat-Tree for *x*500 benchmarks, we observe the opposite in Figure 7 when concurrently running multiple applications. Here, it results in the lowest number (14) of executed runs for MILC, probably due to inter-job interference, and overall the combination of "HyperX / DFSSSP / linear" yields the highest number of finished jobs, outperforming the Fat-Tree by 12.7%, followed by PARX.

The "Fat-Tree / SSSP / clustered" combination appears to be the least effective. However, the drop in completed runs can be (almost entirely) attributed to IMB's Multi-PingPong (MuPP) and the Graph500 benchmark, which seem particularly sensitive to the clustered allocation scheme. When comparing the HyperX with linear and clustered, we do not see this sensitivity in these two

applications, but instead it is visible for the MILC benchmark. This drop in runs for MILC, also visible for "HyperX / DFSSSP / random", can be attributed to both placement and routing. Referring back to Figure 6d, we already saw slowdown and high runtime variability issues for the 32-node case while running MILC in isolation.

The random placement (together with static, shortest-path routing) on our HyperX is the least effective method among our mitigation strategies, as introduced in Section 3. This method only outperformed the alternatives for Netgauge's effective bisection bandwidth test (cf. Figure 5c) and the Graph500 benchmark, see Figure 6l. Similarly, in the multi-application environment the benefit is limited to the MuPP benchmark, by yielding 16% more runs compared to the linear allocation. Hence, based on our data, future research research may be needed: (1) to analyze and mitigate the slowdown of MILC, and (2) to determine which allocation scheme is the best match to an adaptive routing for HyperX topologies.

Overall, instead of the expected slowdowns for all applications on a statically routed HyperX, we see more evidence for the competitiveness of the HyperX topology (vs. Fat-Tree), presumably fostered by its structure, the shortest-path routing and dense placement. The resulting isolation for smaller job sizes may reduce inter-job interference more than it actually creates congestion due to adverse (or worst case) traffic patterns, as discussed in Sections 1 and 2.

6 RELATED WORK

The theoretical construct of Fat-Trees was improved over time, from single-rooted trees [46], over multi-rooted k-ary n-trees [66], to eXtended Generalized Fat-Trees (XGFT) [61] and Real-Life or Parallel-Port Fat-Trees (RLFT / PPFT) [86]. Other serious contenders to connect modern and future supercomputers and data-centers are the Dragonfly [41], deployed in different shapes, e.g., IBM's Blue Waters PERCS network [5], or Cray's Aries network for Theta [65]. Alternatives to these low-diameter and adaptively routed topologies are mostly torus-based, e.g., Pleiades' HyperCube topology [38], the 5D BlueGene/Q networks [11] or the 6D Tofu used in K [4].

Further topologies have been proposed, but only studied theoretically. The Slimfly [9, 81], Express-Mesh [37], and semi-random Skywalk [22] are just a few of them. Usually, cost prohibits large network test beds. Therefore researchers utilize simulations [36, 48, 82], smaller setups [21, 80], use different systems to compare topologies [38], or rely on inexpensive hardware, such as Raspberry Pi [39], to explore various topologies, network sizes and deployment configurations, e.g., tapering of the tree, multi-plane setups, and/or routing options, etc. However, such setups might have too idealistic assumptions for simulations, overlook network-related issues which only manifest at scale, or lack accuracy because of either architectural variations or an unrepresentative compute-to-network performance ratio. To the best of our knowledge, we are first to rewire a large-scale HPC system to perform a true 1-to-1 comparison between a state-of-the-art and an experimental topology.

The realistic choice for HyperX are adaptive routings, such as Valiant's algorithm (VAL) or Universal Global Adaptive Load-balancing (UGAL) [2], or the Dimensionally-Adaptive, Load-balanced (DAL) algorithm [3]. For deterministically routed networks (e.g., InfiniBand fabrics), only a few topology-agnostic options exist which satisfy the deadlock-freedom criterion, such as DFSSSP or SAR [14, 17], LASH [75], or Nue routing [16]. Our PARX routing is the first communication- and topology-aware alternative for 2D HyperX topologies, utilizing IB's multi-pathing feature through LMC. Similar approaches, to avoid hot spots [80] — but not to intentionally take longer paths — or to avoid deadlock-freedom [49] exist. Other proposals to exploit multi-pathing [50], to introduce pattern-awareness [70], or mitigate existing bottlenecks [76], are either not compliant with IB specifications, or do not preemptively adjusting to communication demands, like our PARX routing.

7 CONCLUSION

The high-speed interconnection network, connecting the compute nodes of modern supercomputers, plays a crucial role in achieving scalability and throughput for scientific applications. We build a large-scale, 672-node HPC system with dual-plane interconnect, one using a 3-level Fat-Tree topology and the other using a 12x8 2D HyperX, from the remains of a decommissioned system.

We applied a broad set of MPI and HPC benchmarks, and proxy-application for real HPC workloads — usually used during system procurement — to stress both network topologies in isolated scalability and shared capacity runs. The collected data from our 1-to-1 comparison implies that even a HyperX topology with roughly half-bisection bandwidth, and hence drastically reduced deployment costs, can compete with our 18-ary 3-tree, which theoretically offers more than full-bisection due to the reduced node count at the leafs. This result is even more astonishing, considering that we only had deprecated IB equipment (QDR type) available, which does not feature the required adaptive routing for the HyperX.

We investigated two strategies: MPI rank placement and our novel PARX routing, to circumvent the bottleneck arising from applying a shortest-path, static routing to a HyperX. This PARX prototype shows potential, but will be replaced by true adaptive routing in future HyperX deployments, yielding even better results than ours. Nevertheless, we look forward to seeing that our intentional use of minimal paths and detours in static routings and/or the optimization for specific traffic patterns will be reused.

Our evaluation toolchain, including PARX and collected data, is readily available under **https://gitlab.com/domke/t2hx** for download, giving other researchers the option to reuse and adapt our routing approach or to perform similar studies.

ACKNOWLEDGMENTS

We would like to thank all of our 40 university students and volunteers who were essential in the rewiring process. Moreover, we would like to thank Hewlett Packard Enterprise for funding this project, and Fujitsu for donating a few decommissioned IB switches. The work was supported by JSPS KAKENHI Grant Number JP19H04119 and JST CREST Grant Number JPMJCR1687.

J.D. designed the study, modified the HPC system and software stack, performed the evaluation, analyzed the data, and supervised its execution together with S.Ma., while all authors contributed to writing and editing. T.Y., Y.T., I.I., A.N., and S.Mi. assisted in the system modification and benchmark preparation and execution. N.M., D.F., and N.D. provided technical support and insight into the system modification, as well as topology and routing design.

REFERENCES

[1] Ahmed H. Abdel-Gawad, Mithuna Thottethodi, and Abhinav Bhatele. 2014. RAHTM: Routing Algorithm Aware Hierarchical Task Mapping. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 325–335. https://doi.org/10.1109/SC.2014.32

[2] Dennis Abts and John Kim. 2011. *High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities*. Morgan & Claypool Publishers. http://dx.doi.org/10.2200/S00341ED1V01Y201103CAC014

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. 2009. 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)*. ACM, New York, NY, USA, 41:1–41:11. https://doi.org/10.1145/1654059.1654101

[4] Yuichiro Ajima, Tomohiro Inoue, Shinya Hiramoto, Toshiyuki Shimizu, and Yuzo Takagi. 2012. The Tofu Interconnect. *IEEE Micro* 32, 1 (2012), 21–31. https://doi.org/10.1109/MM.2011.98

[5] Baba Arimilli, Ravi Arimilli, Vicente Chung, Scott Clark, Wolfgang Denzel, Ben Drerup, Torsten Hoefler, Jody Joyner, Jerry Lewis, Jian Li, Nan Ni, and Ram Rajamony. 2010. The PERCS High-Performance Interconnect. In *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*. 75–82. https://doi.org/10.1109/HOTI.2010.16

[6] Sadoon Azizi, Farshad Safaei, and Naser Hashemi. 2013. On the topological properties of HyperX. *The Journal of Supercomputing* 66, 1 (Oct. 2013), 572–593. https://doi.org/10.1007/s11227-013-0935-6

[7] Baidu, Inc. 2017. baidu-allreduce. https://github.com/baidu-research/baidu-allreduce

[8] Claude Bernard, Tom Burch, Thomas A. DeGrand, Carleton DeTar, Steven Gottlieb, Urs M. Heller, James E. Hetrick, Kostas Orginos, Bob Sugar, and Doug Toussaint. 2000. Scaling tests of the improved Kogut-Susskind quark action. *Physical Review D* 61, 11 (April 2000), 4. https://doi.org/10.1103/PhysRevD.61.111502

[9] Maciej Besta and Torsten Hoefler. 2014. 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)*. IEEE Press, Piscataway, NJ, USA, 348–359. https://doi.org/10.1109/SC.2014.34

[10] Kevin Brown, Jens Domke, and Satoshi Matsuoka. 2015. Hardware-Centric Analysis of Network Performance for MPI Applications. In *2015 21th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Press, Melbourne, Australia, 8.

[11] Dong Chen, Noel Eisley, Philip Heidelberger, Robert Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David Satterfield, Burkhard Steinmacher-Burow, and Jeffrey Parker. 2012. The IBM Blue Gene/Q Interconnection Fabric. *IEEE Micro* 32, 1 (Jan. 2012), 32–43. https://doi.org/10.1109/MM.2011.96

[12] Charles Clos. 1953. A Study of Non-Blocking Switching Networks. *The Bell System Technical Journal* 32, 2 (March 1953), 406–424. https://doi.org/10.1002/j.1538-7305.1953.tb01433.x

[13] William Dally and Brian Towles. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] Jens Domke and Torsten Hoefler. 2016. Scheduling-Aware Routing for Supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, 13:1–13:12. http://dl.acm.org/citation.cfm?id=3014904.3014922

[15] Jens Domke, Torsten Hoefler, and Satoshi Matsuoka. 2014. 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)*. IEEE Press, New Orleans, LA, USA, 597–608. https://doi.org/10.1109/SC.2014.54

[16] Jens Domke, Torsten Hoefler, and Satoshi Matsuoka. 2016. 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)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2907294.2907313

[17] Jens Domke, Torsten Hoefler, and Wolfgang E. Nagel. 2011. Deadlock-Free Oblivious Routing for Arbitrary Topologies. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Washington, DC, USA, 613–624.

[18] Jack Dongarra, Michael Heroux, and Piotr Luszczek. 2015. *HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems*. Technical Report ut-eecs-15-736. University of Tennessee. https://library.eecs.utk.edu/pub/594

[19] Exascale Computing Project. 2018. ECP Proxy Apps Suite. https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/

[20] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. 2012. 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)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 103:1–103:9. http://dl.acm.org/citation.cfm?id=2388996.2389136

[21] Samuel A. Fineberg and Kevin T. Pedretti. 1999. Analysis of 100Mb/s Ethernet for the Whitney Commodity Computing Testbed. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '99)*. IEEE Computer Society, Washington, DC, USA, 276–. http://dl.acm.org/citation.cfm?id=795668.796738

[22] Ikki Fujiwara, Michihiro Koibuchi, Hiroki Matsutani, and Henri Casanova. 2014. Skywalk: A Topology for HPC Networks with Low-Delay Switches. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 263–272. https://doi.org/10.1109/IPDPS.2014.37

[23] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.

[24] GSIC, Tokyo Institute of Technology. 2013. TSUBAME2.5 Hardware and Software. https://www.gsic.titech.ac.jp/sites/default/files/spec25e1.pdf

[25] Francois Gygi. 2008. Architecture of Qbox: A Scalable First-principles Molecular Dynamics Code. *IBM Journal of Research and Development* 52, 1/2 (Jan. 2008), 137–144. http://dl.acm.org/citation.cfm?id=1375990.1376003

[26] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Venkatram Vishwanath, Tom Peterka, Joe Insley, David Daniel, Patricia Fasel, and Zarija Lukić. 2016. HACC: Extreme Scaling and Performance Across Diverse Architectures. *Commun. ACM* 60, 1 (Dec. 2016), 97–104. https://doi.org/10.1145/3015569

[27] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.

[28] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Austin, TX, USA, 73:1–73:12.

[29] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Netgauge: A Network Performance Measurement Framework. In *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, Vol. 4782. Springer, 659–671.

[30] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2008. 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.

[31] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2009. Optimized Routing for Large-Scale InfiniBand Networks. In *17th Annual IEEE Symposium on High Performance Interconnects (HOTI 2009)*.

[32] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.12

[33] Torsten Hoefler and Marc Snir. 2011. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, Tucson, AZ, 75–85.

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

[35] Intel Corporation. 2018. Intel® MPI Benchmarks User Guide. https://software.intel.com/en-us/imb-user-guide

[36] Nikhil Jain, Abhinav Bhatele, Louis H. Howell, David Böhme, Ian Karlin, Edgar A. León, Misbah Mubarak, Noah Wolfe, Todd Gamblin, and Matthew L. Leininger. 2017. Predicting the Performance Impact of Different Fat-tree Configurations. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, 50:1–50:13. https://doi.org/10.1145/3126908.3126967 Note: LLNL-CONF-736289.

[37] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Todd Gamblin, and Laxmikant V. Kalé. 2017. Partitioning Low-Diameter Networks to Eliminate Inter-Job Interference. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017 (IPDPS '17)*. IEEE Computer Society, 439–448. https://doi.org/10.1109/IPDPS.2017.91

[38] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. 2011. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Comput.* 37, 9 (2011), 562–575. https://doi.org/10.1016/j.parco.2011.02.002

[39] Steven J. Johnston, Philip J. Basford, Colin S. Perkins, Herry Herry, Fung Po Tso, Dimitrios Pezaros, Robert D. Mullins, Eiko Yoneki, Simon J. Cox, and Jeremy Singer. 2018. Commodity single board computer clusters and their applications. *Future Generation Computer Systems* 89 (2018), 201–212. https://doi.org/10.1016/j.future.2018.06.048

[40] Georgios Kathareios, Cyriel Minkenberg, Bogdan Prisacari, German Rodriguez, and Torsten Hoefler. 2015. Cost-effective Diameter-two Topologies: Analysis and Evaluation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, 36:1–36:11. https://doi.org/10.1145/2807591.2807652

[41] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. *ACM SIGARCH Comput. Architecture News* 36, 3 (June 2008), 77–88. https://doi.org/10.1145/1394608.1382129

[42] Benjamin Klenk and Holger Fröning. 2017. An Overview of MPI Characteristics of Exascale Proxy Applications. In *High Performance Computing: 32nd International Conference, ISC High Performance 2017 (ISC '17)*. Frankfurt, Germany, 217–236. https://doi.org/10.1007/978-3-319-58667-0_12

[43] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. 2008. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz (Eds.). Springer Berlin Heidelberg, 139–155.

[44] Lawrence Livermore National Laboratory. 2019. Qbox: Computing Electronic Structures at the Quantum Level. https://computation.llnl.gov/projects/qbox-computing-structures-quantum-level

[45] Matt Leininger. 2014. CORAL Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/

[46] Charles E. Leiserson. 1985. Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Trans. Comput.* 34, 10 (Oct. 1985), 892–901. http://dl.acm.org/citation.cfm?id=4492.4495

[47] Edgar A. León, Ian Karlin, Abhinav Bhatele, Steven H. Langer, Chris Chambreau, Louis H. Howell, Trent D'Hooge, and Matthew L. Leininger. 2016. 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)*. IEEE Press, Piscataway, NJ, USA, 78:1–78:12. http://dl.acm.org/citation.cfm?id=3014904.3015009

[48] Ning Liu, Adnan Haider, Xian-He Sun, and Dong Jin. 2015. 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)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/2769458.2769474

[49] Pedro López, José Flich, and José Duato. 2001. Deadlock-free routing in InfiniBand through destination renaming. In *International Conference on Parallel Processing, 2001. (ICPP '01)*. 427–434. https://doi.org/10.1109/ICPP.2001.952089

[50] J. C. Martínez, José Flich, Antonio Robles, Pedro López, and José Duato. 2003. Supporting Fully Adaptive Routing in InfiniBand Networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. IEEE Computer Society, Washington, DC, USA, 44.1–54.1. http://dl.acm.org/citation.cfm?id=838237.838493

[51] Message Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard Version 3.1. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[52] George Michelogiannakis, Khaled Z. Ibrahim, John Shalf, Jeremiah J. Wilke, Samuel Knight, and Joseph P. Kenny. 2017. APHiD: Hierarchical Task Placement to Enable a Tapered Fat Tree Topology for Lower Power and Cost in HPC Networks. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17)*. IEEE Press, Piscataway, NJ, USA, 228–237. https://doi.org/10.1109/CCGRID.2017.33

[53] Takahiro Misawa, Satoshi Morita, Kazuyoshi Yoshimi, Mitsuaki Kawamura, Yuichi Motoyama, Kota Ido, Takahiro Ohgoe, Masatoshi Imada, and Takeo Kato. 2018. mVMC–Open-source software for many-variable variational Monte Carlo method. *Computer Physics Communications* (2018). https://doi.org/10.1016/j.cpc.2018.08.014

[54] Jamaludin Mohd-Yusof, Sriram Swaminarayan, and Timothy C. Germann. 2013. *Co-design for molecular dynamics: An exascale proxy application.* Technical Report LA-UR 13-20839. Los Alamos National Laboratory. http://www.lanl.gov/orgs/adtsc/publications/science_highlights_2013/docs/Pg88_89.pdf

[55] Misbah Mubarak, Nikhil Jain, Jens Domke, Noah Wolfe, Caitlin Ross, Kelvin Li, Abhinav Bhatele, Christopher D. Carothers, Kwan-Liu Ma, and Robert B. Ross. 2017. Toward Reliable Validation of HPC Interconnect Simulations. In *Proceedings of the 2017 Winter Simulation Conference (WSC '17)*. IEEE Press, Las Vegas, NV, USA, 659–674.

[56] Jayaram Mudigonda, Praveen Yalagandula, and Jeffrey C. Mogul. 2011. Taming the Flying Cable Monster: A Topology Design and Optimization Framework for Data-center Networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'11)*. USENIX Association, Berkeley, CA, USA, 14. http://dl.acm.org/citation.cfm?id=2002181.2002189

[57] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the Graph 500. In *Proceedings of the Cray User's Group Meeting (CUG)*. 5.

[58] Takahito Nakajima, Michio Katouda, Muneaki Kamiya, and Yutaka Nakatsuka. 2014. NTChem: A High-Performance Software Package for Quantum Molecular Simulation. *International Journal of Quantum Chemistry* 115, 5 (Dec. 2014), 349–359. https://doi.org/10.1002/qua.24860

[59] National Energy Research Scientific Computing Center (NERSC). 2016. NERSC-8 / Trinity Benchmarks. https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/

[60] National Energy Research Scientific Computing Center (NERSC). 2018. MILC. https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/milc/

[61] Sabine R. Öhring, Maximilian Ibel, Sajal K. Das, and Mohan J. Kumar. 1995. On generalized fat trees. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 37.

[62] Kenji Ono, Masako Iwata, Tsuyoshi Tamaki, Yasuhiro Kawashima, Kei Akasaka, Soichiro Suzuki, Junya Onishi, Ken Uzawa, Kazuhiro Hamaguchi, Yohei Miyazaki, and Masashi Imano. [n. d.]. FFV-C package. http://avr-aics-riken.github.io/ffvc_package/

[63] OpenFabrics Alliance. 2018. OpenSM. https://github.com/linux-rdma/opensm

[64] Jongsoo Park, Mikhail Smelyanskiy, Ulrike Meier Yang, Dheevatsa Mudigere, and Pradeep Dubey. 2015. High-performance Algebraic Multigrid Solver Optimized for Multi-core Based Distributed Parallel Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, Austin, TX, USA, 54:1–54:12. https://doi.org/10.1145/2807591.2807603

[65] Scott Parker, Vitali Morozov, Sudheer Chunduri, Kevin Harms, Chris Knight, and Kalyan Kumaran. 2017. Early Evaluation of the Cray XC40 Xeon Phi System 'Theta' at Argonne. In *Proceedings of the Cray User's Group Meeting (CUG) (CUG '17)*. https://www.osti.gov/biblio/1393541

[66] Fabrizio Petrini and Marco Vanneschi. 1997. k-ary n-trees: high performance networks for massively parallel architectures. In *11th International Parallel Processing Symposium*. 87–93. https://doi.org/10.1109/IPPS.1997.580853

[67] Samuel D. Pollard, Nikhil Jain, Stephen Herbein, and Abhinav Bhatele. 2018. Evaluation of an Interference-free Node Allocation Policy on Fat-tree Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, 26:1–26:13. http://dl.acm.org/citation.cfm?id=3291656.3291691

[68] RIKEN AICS. 2015. Fiber Miniapp Suite. https://fiber-miniapp.github.io/

[69] Gonzalo P. Rodrigo Álvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. 2015. HPC System Lifetime Story: Workload Characterization and Evolutionary Analyses on NERSC Systems. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, New York, NY, USA, 57–60. https://doi.org/10.1145/2749246.2749270

[70] German Rodriguez, Ramon Beivide, Cyriel Minkenberg, Jesus Labarta, and Mateo Valero. 2009. Exploring Pattern-aware Routing in Generalized Fat Tree Networks. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 276–285. https://doi.org/10.1145/1542275.1542316

[71] SchedMD LLC. 2019. slurm.conf (SelectType). https://slurm.schedmd.com/slurm.conf.html

[72] Michael D. Schroeder, Andrew D. Birell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. 1991. Autonet: A High-speed, Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communications* 9, 8 (Oct. 1991).

[73] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications* 20 (2006), 287–331.

[74] Alexander Shpiner, Zachy Haramaty, Saar Eliad, Vladimir Zdornov, Barak Gafni, and Eitan Zahavi. 2017. Dragonfly+: Low Cost Topology for Scaling Datacenters. In *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*. 1–8. https://doi.org/10.1109/HiPINEB.2017.11

[75] Tor Skeie, Olav Lysne, and Ingebjørg Theiss. 2002. Layered Shortest Path (LASH) Routing in Irregular System Area Networks. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Washington, DC, USA, 194.

[76] Staci A. Smith, Clara E. Cromey, David K. Lowenthal, Jens Domke, Nikhil Jain, Jayaraman J. Thiagarajan, and Abhinav Bhatele. 2018. Mitigating Inter-Job Interference Using Adaptive Flow-Aware Routing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, 27:1–27:15. http://dl.acm.org/citation.cfm?id=3291656.3291692

[77] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. 2018. TOP500. http://www.top500.org/

[78] Stanimire Tomov, Azzam Haidar, Daniel Schultz, and Jack Dongarra. 2018. *Evaluation and Design of FFT for Distributed Accelerated Systems.* Tech Report FFT-ECP ST-MS-10-1216. Innovative Computing Laboratory, University of Tennessee. https://www.icl.utk.edu/files/publications/2018/icl-utk-1079-2018.pdf

[79] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. 2016. Extreme scale breadth-first search on supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*. 1040–1047. https://doi.org/10.1109/BigData.2016.7840705

[80] Abhinav Vishnu, Matt Koop, Adam Moody, Amith R. Mamidala, Sundeep Narravula, and Dhabaleswar K. Panda. 2007. Hot-Spot Avoidance With Multi-Pathing Over InfiniBand: An MPI Perspective. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. 479–486. https://doi.org/10.1109/CCGRID.2007.60

[81] Noah Wolfe, Christopher D. Carothers, Misbah Mubarak, Robert Ross, and Philip Carns. 2016. 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)*. ACM, New York, NY, USA, 189–199. https://doi.org/10.1145/2901378.2901389

[82] Noah Wolfe, Misbah Mubarak, Nikhil Jain, Jens Domke, Abhinav Bhatele, Christopher D. Carothers, and Robert B. Ross. 2017. Preliminary Performance Analysis of Multi-rail Fat-tree Networks. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17)*. IEEE Press, Madrid, Spain, 258–261. https://doi.org/10.1109/CCGRID.2017.102 Short paper.

[83] Xu Yang, John Jenkins, Misbah Mubarak, Robert B. Ross, and Zhiling Lan. 2016. Watch out for the Bully!: Job Interference Study on Dragonfly Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, 64:1–64:11. http://dl.acm.org/citation.cfm?id=3014904.3014990

[84] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. 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*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60. http://dx.doi.org/10.1007/10968987_3

[85] Eitan Zahavi. 2010. *D-Mod-K Routing Providing Non-Blocking Traffic for Shift Permutations on Real Life Fat Trees*. Technical Report. https://webee.technion.ac.il/publication-link/index/id/574

[86] Eitan Zahavi. 2012. Fat-tree Routing and Node Ordering Providing Contention Free Traffic for MPI Global Collectives. *J. Parallel Distrib. Comput.* 72, 11 (Nov. 2012), 1423–1432. https://doi.org/10.1016/j.jpdc.2012.01.018

A ARTIFACT DESCRIPTION

SUMMARY OF THE EXPERIMENTS REPORTED

We developed a framework of scripts and git submodules to manage the R&D of the routing engine, to set up the benchmarking infrastructure, and to perform the measurements. After cloning our repository https://gitlab.com/domke/t2hx (or downloading the artifacts from https://doi.org/10.5281/zenodo.3375075), one has access to all benchmarks (see Table 2), patches, and scripts. Only minor modifications to the configuration files should be necessary, such as changing host names, or adjustments to different node counts for the benchmarks, before testing on another system. If users deviate from our OS version (CentOS Linux release 7.4.1708) then some additional changes might be required.

Cloning instruction for framework:

- git clone –recurse-submodules https://gitlab.com/domke/t2hx

Required access rights:

- requires **root** level access when changing InfiniBand routing with installed subnet manager
- maybe **root** when interfacing with the admin node which runs our novel PARX routing
- user level to benchmark with other/existing routing engines

OS-level dependencies (based on our CentOS 7.4):

- cmake autoconf automake libtool cpupowerutils screen pdsh systemd-devel libnl3-devel valgrind-devel bison flex gcc gcc-c++ gcc-gfortran wget libudev-devel zlib-devel libstdc++-devel pciutils tcl tcl-devel tk glib2-devel kernel-devel vim rpm-build pkgconfig make python2-numpy numactl-devel lsof psmisc git swig python-devel python2-clustershell rdma-core-devel

HowTo use this framework (follow these instructions):

Installation:

- `./inst/_init.sh routing` on a admin node, which will run OpenSM with PARX (req. **root**)
- `./inst/_init.sh` on one compute node with access to parallel FS or NFS

Generation of host lists for MPI:

- modify and/or re-run `./inst/_gethostlists.sh`

Compile each benchmark:

- execute `./inst/*.sh` for all files (except those starting with underscore)

Adjust configuration if necessary:

- change path to Intel compiler: `conf/intel.cfg`
- change settings, such as input/#nodes/etc., per benchmark if desired: `conf/*.cfg`
- select topology/routing/placement to test different network: `conf/t2hx.sh`
- change `RUNMODE` when switching from capability to capacity runs: `conf/t2hx.sh`
- change `NumOMP`, `HXSMHOST`, `OSM0TRIGGER`, etc., if needed in file: `conf/t2hx.sh`

Running all benchmarks:

- execute `./run/*.sh` for every benchmark

A few hints for working with the framework:

(1) if cloned directory is not in home/NFS or parallel FS, then additional steps will be needed
(2) install scripts set benchmark version (via git commit hash) to exact version used in the paper
(3) other versioning information for libraries, kernel, etc. will be listed in appendix AE
(4) setup assumes `mlx4_*` naming of HCA, and 1st HCA is attached to Fat-Tree, and 2nd to HyperX
(5) further changes may be needed (to `conf/*` and `run/*`) if the statement above is not true

Explanation of relevant directory structure:

`./`

- contains sub-directories for each benchmark

`conf/`

- configuration files for benchmarks (inputs, node counts, paths, etc) and host files

`conf/comm4parx.tgz`

- contains communication matrices for all benchmarks (and sizes) needed as inputs for PARX
- extracts into `conf/comm4parx/` sub-directory

`conf/hostfiles.tgz`

- our host files for linear, clustered, and random placement (as used for the paper)
- extracts files directly into `conf/`

`conf/opensm/`

- OpenSM configurations for Fat-Tree (ftree/sssp routing), and HyperX (dfsssp/parx routing)
- guid files of HCAs as required by routings (adjust to system)
- file `*.guid2lid.cfg` must be changed to assign LIDs/GUIDs to correct quadrants for PARX
- LID policy for quadrants: $Q_0 := 0 \ldots 999$, $Q_1 := 1000 \ldots 1999$, $Q_2 := 2000 \ldots 2999$, $Q_3 := 3000 \ldots 3999$
- LID policy for switches in quadrants: see above but add 10000

`dep/`

- multiple auxiliary scripts for managing, pre-/post-processing the measurements
- additional library dependencies, e.g. OpenMPI (automatically compiled by framework)

`inst/`

- scripts to install/compile dependencies and benchmarks

`log/`

- directory will contain outputs of the benchmarks

`log/*.tbz2`

- all outputs of our benchmarking for the paper; extracts into per-BM sub-directory

`patches/`

- req. patches for: benchmark timing, OpenSM, OpenMPI, etc.

`run/`

- scripts to execute the benchmarks

`paper/`

- processed logs, tex files, figures, gnuplot stuff, etc.

ARTIFACT AVAILABILITY

Software Artifact Availability: (One of these options remains.) All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: (One of these options remains.) There are no author-created hardware artifacts.

Data Artifact Availability: (One of these options remains.) All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: (One of these options remains.) No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

`https://doi.org/10.5281/zenodo.3375075`
`https://gitlab.com/domke/t2hx`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Hewlett-Packard HP ProLiant SL390s G7 compute nodes, Intel Xeon X5670 CPUs, NVIDIA Tesla K20X GPUs, QDR InfiniBand HCAs, Voltaire Grid Director 4700, Voltaire Grid Director 4036, 3-level full-bisection bandwidth Fat-Tree and 12x8 2D HyperX network topology

Operating systems and versions: CentOS Linux release 7.4.1708 running kernel 3.10.0-693.21.1

Compilers and versions: Gnu Fortran/C/C++ 4.8.5 and Intel Parallel Studio XE (version 2018; update 3)

Applications and versions: ECP Proxy-Apps (AMG, CoMD, miniFE, SWFFT), RIKEN R-CCS Fiber Miniapps (FFVC, mVMC, NTChem), Trinity benchmark (MILC), CORAL benchmark (qb@ll), Graph500, HPL, HPCG, Intel MPI Benchmarks, Netgauge, mpiGraph, Baidu DeepBench [all versions tagged by git commit hashes or download versions]

Libraries and versions: OpenMPI v1.10.7, OFED v4.8-2, OpenSM v3.3.21, FFTW v3.3.4, LAPACK v3.8.0, ScaLAPACK v2.0.2, BLAS v3.8.0, Xerces-C v3.2.2, METIS v5.1.0, libcsv v3.0.3, TAU v2.27, ibprof v0, OTF v1.12.5

Key algorithms: ftree, SSSP, DFSSSP, PARX

Input datasets and versions: as provided by applications and benchmarks

Paper Modifications: Modifications to applications and benchmarks are as follows:

(1) add walltime measurement around main solver or kernel of the application benchmarks;
(2) remove CUDA dependency from Baidu's Allreduce and reduce iteration counter for some sizes;
(3) allow IMB to test very large messages for collectives and add SleepyAllreduce test case;
(4) patch issue in mVMC to not call MPI inside of OMP regions; and
(5) fix compilation bug in qb@ll.

Modifications to OpenSM are as follows:

(1) add PARX routing to read and adapt to communication-demand matrix; and
(2) added re-routing trigger mechnism to semi-automatically run PARX for changing demands.

Modifications to OpenMPI are as follows:

(1) fix bug in service level (SL) inquiry from OpenSM as required for VL-based deadlock-freedom;
(2) changed BFO PML to choose correct path based on message size and HyperX quadrant; and
(3) add PERUSE/ibprof support to collect low level point-to-point data needed for PARX routing.

Output from scripts that gathers execution environment information.

```
       LSB Version:        :core-4.1-amd64:core-4.1⌋
       ↪ -noarch
Distributor ID:        CentOS
Description:        CentOS Linux release 7.4.1708
↪ (Core)
Release:        7.4.1708
Codename:        Core
Linux r60n00.mng.t2.gsic.titech.ac.jp
↪ 3.10.0-693.21.1.el7.x86_64 #1 SMP Wed Mar 7
↪ 19:03:37 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU
↪ X5670 @ 2.93GHz
Stepping:              2
CPU MHz:               1596.000
CPU max MHz:           2927.0000
CPU min MHz:           1596.0000
BogoMIPS:              5867.31
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23
```

```
Flags:                 fpu vme de pse tsc msr pae mce
↪  cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↪  dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
↪  pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
↪  bts rep_good nopl xtopology nonstop_tsc
↪  aperfmperf pni pclmulqdq dtes64 monitor ds_cpl
↪  vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca
↪  sse4_1 sse4_2 popcnt aes lahf_lm epb spec_ctrl
↪  ibpb_support tpr_shadow vnmi flexpriority ept
↪  vpid dtherm ida arat
MemTotal:       55641952 kB
MemFree:        54156712 kB
MemAvailable:   54312284 kB
Buffers:               0 kB
Cached:           609708 kB
SwapCached:            0 kB
Active:           148016 kB
Inactive:         485328 kB
Active(anon):      27248 kB
Inactive(anon):   120556 kB
Active(file):     120768 kB
Inactive(file):   364772 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:         23652 kB
Mapped:            19524 kB
Shmem:            124152 kB
Slab:             106188 kB
SReclaimable:      43388 kB
SUnreclaim:        62800 kB
KernelStack:        5280 kB
PageTables:         4172 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    27820976 kB
Committed_AS:     211348 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      282396 kB
VmallocChunk:   34327506940 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      207036 kB
DirectMap2M:     6074368 kB
DirectMap1G:    50331648 kB
XDG_SESSION_ID=162
HOSTNAME=r60n00.mng.t2.gsic.titech.ac.jp
TERM=screen
SHELL=/bin/bash
HISTSIZE=1000
USER=root
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=0
↪  1;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;0
↪  1:mi=01;05;37;41:su=37;41:sg=30;43:ca=30;41:tw=3
↪  0;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tg
↪  z=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lh
↪  a=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.t
↪  lz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.z
↪  ip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;
↪  31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31
↪  :*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31
↪  :*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:
↪  *.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:
↪  *.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31
↪  :*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*
↪  .jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:
↪  *.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:
↪  *.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35
↪  :*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;3
↪  5:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;
↪  35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01
↪  ;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01
↪  ;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;
↪  35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01
↪  ;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;3
↪  5:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;3
↪  5:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;3
↪  5:*.aac=01;36:*.au=01;36:*.flac=01;36:*.mid=01;3
↪  6:*.midi=01;36:*.mka=01;36:*.mp3=01;36:*.mpc=01;
↪  36:*.ogg=01;36:*.ra=01;36:*.wav=01;36:*.axa=01;3
↪  6:*.oga=01;36:*.spx=01;36:*.xspf=01;36:
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/
↪  sbin:/usr/bin:/root/bin:/home/usr0/domke-j-aa/in
↪  xi-3.0.32-1:/home/usr0/domke-j-aa/lshw-B.02.18/s
↪  rc
MAIL=/var/spool/mail/root
PWD=/home/usr0/domke-j-aa/t2hx/paper/data/sc-author-
↪  kit
HISTCONTROL=ignoredups
HOME=/root
SHLVL=2
LOGNAME=root
LESSOPEN=||/usr/bin/lesspipe.sh %s
_=/bin/env
System:    Host: r60n00.mng.t2.gsic.titech.ac.jp
↪  Kernel: 3.10.0-693.21.1.el7.x86_64 x86_64 bits:
↪  64 Console: N/A
          Distro: CentOS Linux release 7.4.1708 (Core)
Machine:   Type: Multimount-chassis System: HP
↪  product: ProLiant SL390s G7 v: N/A serial:
↪  JPT0333WTP
           Mobo: HP model: N/A serial: JPT0333WTP
            ↪  BIOS: HP v: P69 date: 05/21/2018
```

```
CPU:       Topology: 2x 6-Core model: Intel Xeon X5670
↪  bits: 64 type: MT MCP SMP L2 cache: 24.0 MiB
           Speed: 1862 MHz min/max: 1596/2927 MHz Core
           ↪  speeds (MHz): 1: 2927 2: 1596 3: 1729
           ↪  4: 1862 5: 1596 6: 2394
           7: 1729 8: 1729 9: 1729 10: 1596 11: 1596
           ↪  12: 1596 13: 1596 14: 1596 15: 1596 16:
           ↪  1596 17: 1596 18: 1596
           19: 1596 20: 1729 21: 1729 22: 1596 23:
           ↪  1596 24: 1729
Graphics:  Device-1: Advanced Micro Devices [AMD/ATI]
↪  ES1000 driver: radeon v: kernel
           Device-2: NVIDIA GK110GL [Tesla K20Xm]
           ↪  driver: nouveau v: kernel
           Device-3: NVIDIA GK110GL [Tesla K20Xm]
           ↪  driver: nouveau v: kernel
           Device-4: NVIDIA GK110GL [Tesla K20Xm]
           ↪  driver: nouveau v: kernel
           Display: server: No display server data
           ↪  found. Headless machine? tty: 126x20
           Message: Unable to show advanced data.
           ↪  Required tool glxinfo missing.
Audio:     Message: No Device data found.
Network:   Device-1: Intel 82576 Gigabit Network
↪  driver: igb
           IF: eth0 state: up speed: 1000 Mbps duplex:
           ↪  full mac: 78:e7:d1:20:fb:d0
           Device-2: Intel 82576 Gigabit Network
           ↪  driver: igb
           IF: eth1 state: down mac: 78:e7:d1:20:fb:d1
           Device-3: Mellanox MT26438 [ConnectX VPI
           ↪  PCIe 2.0 5GT/s - IB QDR / 10GigE
           ↪  Virtualization+] driver: mlx4_core
           IF: eth2 state: down mac: 78:e7:d1:20:fb:d6
           Device-4: Mellanox MT26428 [ConnectX VPI
           ↪  PCIe 2.0 5GT/s - IB QDR / 10GigE]
           ↪  driver: mlx4_core
           IF: ib0 state: down mac:
           ↪  80:00:02:08:fe:80:00:00:00:00:00:00:0
           ↪  0:02:c9:03:00:0b:0c:ad
           IF-ID-1: ib1 state: down mac:
           ↪  80:00:02:09:fe:80:00:00:00:00:00:00:0
           ↪  0:02:c9:03:00:0b:0c:ae
           IF-ID-2: ib2 state: down mac:
           ↪  80:00:02:48:fe:81:11:00:00:00:00:00:7
           ↪  8:e7:d1:03:00:20:fb:d5
Drives:    Local Storage: total: 111.80 GiB used: 0
↪  KiB (0.0%)
           ID-1: /dev/sda model: MK0060EAVDR size:
           ↪  55.90 GiB
           ID-2: /dev/sdb model: MK0060EAVDR size:
           ↪  55.90 GiB
Partition: ID-1: / size: 111.74 GiB used: 14.08 GiB
↪  (12.6%) fs: nfs4 remote:
↪  10.4.0.132:/opt/osimages/centos/7.4
Sensors:   Message: No ipmi sensors data was found.
           Missing: Required tool sensors not
           ↪  installed. Check --recommends
Info:      Processes: 289 Uptime: 4d 22h 40m Memory:
↪  53.06 GiB used: 869.8 MiB (1.6%) Init: systemd
↪  runlevel: 5
           Shell: collect_environ inxi: 3.0.32
NAME MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda    8:0    0 55.9G  0 disk
sdb    8:16   0 55.9G  0 disk
[0:0:0:0]    disk    ATA      MK0060EAVDR      HPG6
↪  /dev/sda   60.0GB
[1:0:0:0]    disk    ATA      MK0060EAVDR      HPG6
↪  /dev/sdb   60.0GB
H/W path             Device    Class      Description
=====================================================
                               system     ProLiant SL390s
                                 ↪  G7 (605081-B21)
/0                               bus        Motherboard
/0/0                             memory     64KiB BIOS
/0/400                           processor  Intel(R)
↪  Xeon(R) CPU           X5670  @ 2.93GHz
/0/400/710                       memory     192KiB L1
↪  cache
/0/400/720                       memory     1536KiB L2
↪  cache
/0/400/730                     memory     12MiB L3 cache
/0/406                           processor  Intel(R)
↪  Xeon(R) CPU           X5670  @ 2.93GHz
/0/406/716                       memory     192KiB L1
↪  cache
/0/406/726                       memory     1536KiB L2
↪  cache
/0/406/736                     memory     12MiB L3 cache
/0/1000                          memory     System Memory
/0/1000/0                        memory     4GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1000/1                        memory     4GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1000/2                        memory     4GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1000/3                        memory     4GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1000/4                        memory     4GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1000/5                        memory     4GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1001                        memory     System Memory
/0/1001/0                        memory     2GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1001/1                        memory     8GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1001/2                        memory     2GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
```

```
/0/1001/3                       memory     8GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1001/4                       memory     2GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/1001/5                       memory     8GiB DIMM
↪  DDR3 Synchronous 1333 MHz (0.8 ns)
/0/b                            memory
/0/c                            memory
/0/100                          bridge     5520 I/O Hub
↪  to ESI Port
/0/100/1                       bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 1
/0/100/1/0           eth0       network    82576
↪  Gigabit Network Connection
/0/100/1/0.1         eth1       network    82576
↪  Gigabit Network Connection
/0/100/2                       bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 2
/0/100/3                       bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 3
/0/100/3/0           ib1        network    MT26428
↪  [ConnectX VPI PCIe 2.0 5GT/s - IB QDR / 10GigE]
/0/100/4                       bridge     5520/X58 I/O
↪  Hub PCI Express Root Port 4
/0/100/5                       bridge     5520/X58 I/O
↪  Hub PCI Express Root Port 5
/0/100/5/0           eth2       network    MT26438
↪  [ConnectX VPI PCIe 2.0 5GT/s - IB QDR / 10GigE
↪  Virtualization+]
/0/100/6                       bridge     5520/X58 I/O
↪  Hub PCI Express Root Port 6
/0/100/7                       bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 7
/0/100/7/0                      display    GK110GL
↪  [Tesla K20Xm]
/0/100/8                       bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 8
/0/100/9                        bridge
↪  7500/5520/5500/X58 I/O Hub PCI Express Root Port 9
/0/100/a                        bridge
↪  7500/5520/5500/X58 I/O Hub PCI Express Root Port
↪  10
/0/100/14                       generic
↪  7500/5520/5500/X58 I/O Hub System Management
↪  Registers
/0/100/14.1                     generic
↪  7500/5520/5500/X58 I/O Hub GPIO and Scratch Pad
↪  Registers
/0/100/14.2                     generic
↪  7500/5520/5500/X58 I/O Hub Control Status and RAS
↪  Registers
/0/100/1c                       bridge     82801JI
↪  (ICH10 Family) PCI Express Root Port 1
/0/100/1c.4                     bridge     82801JI
↪  (ICH10 Family) PCI Express Root Port 5
/0/100/1c.4/0                   generic    Integrated
↪  Lights-Out Standard Slave Instrumentation &
↪  System Support
/0/100/1c.4/0.2                 generic    Integrated
↪  Lights-Out Standard Management Processor Support
↪  and Messaging
/0/100/1c.4/0.4                 bus        Integrated
↪  Lights-Out Standard Virtual USB Controller
/0/100/1c.4/0.4/1    usb6       bus        UHCI Host
↪  Controller
/0/100/1c.4/0.4/1/1             input      Virtual
↪  Keyboard
/0/100/1d                       bus        82801JI
↪  (ICH10 Family) USB UHCI Controller #1
/0/100/1d/1          usb2       bus        UHCI Host
↪  Controller
/0/100/1d.1                     bus        82801JI
↪  (ICH10 Family) USB UHCI Controller #2
/0/100/1d.1/1        usb3       bus        UHCI Host
↪  Controller
/0/100/1d.2                     bus        82801JI
↪  (ICH10 Family) USB UHCI Controller #3
/0/100/1d.2/1        usb4       bus        UHCI Host
↪  Controller
/0/100/1d.3                     bus        82801JI
↪  (ICH10 Family) USB UHCI Controller #6
/0/100/1d.3/1        usb5       bus        UHCI Host
↪  Controller
/0/100/1d.7                     bus        82801JI
↪  (ICH10 Family) USB2 EHCI Controller #1
/0/100/1d.7/1        usb1       bus        EHCI Host
↪  Controller
/0/100/1e                       bridge     82801 PCI
↪  Bridge
/0/100/1e/3                     display    ES1000
/0/100/1f                       bridge     82801JIR
↪  (ICH10R) LPC Interface Controller
/0/100/1f.2                     storage    82801JI
↪  (ICH10 Family) SATA AHCI Controller
/0/101                          bridge     Intel
↪  Corporation
/0/102                          bridge     Intel
↪  Corporation
/0/103                          bridge     Intel
↪  Corporation
/0/104                          bridge     Intel
↪  Corporation
/0/105                          bridge
↪  7500/5520/5500/X58 Physical Layer Port 0
/0/106                          bridge
↪  7500/5520/5500 Physical Layer Port 1
/0/107                          bridge     Intel
↪  Corporation
```

```
/0/108                          bridge     Intel
↪  Corporation
/0/109                          bridge     Intel
↪  Corporation
/0/10a                          bridge     Intel
↪  Corporation
/0/10b                          bridge     Intel
↪  Corporation
/0/10c                          bridge     Intel
↪  Corporation
/0/1                           bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 1
/0/2                           bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 2
/0/3                           bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 3
/0/3/0                          display    GK110GL
↪  [Tesla K20Xm]
/0/4                           bridge     5520/X58 I/O
↪  Hub PCI Express Root Port 4
/0/5                           bridge     5520/X58 I/O
↪  Hub PCI Express Root Port 5
/0/6                           bridge     5520/X58 I/O
↪  Hub PCI Express Root Port 6
/0/7                           bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 7
/0/7/0                          display    GK110GL
↪  [Tesla K20Xm]
/0/8                           bridge     5520/5500/X58
↪  I/O Hub PCI Express Root Port 8
/0/9                            bridge
↪  7500/5520/5500/X58 I/O Hub PCI Express Root Port 9
/0/a                            bridge
↪  7500/5520/5500/X58 I/O Hub PCI Express Root Port
↪  10
/0/10d                          bridge     Intel
↪  Corporation
/0/10d/14                       generic
↪  7500/5520/5500/X58 I/O Hub System Management
↪  Registers
/0/10d/14.1                     generic
↪  7500/5520/5500/X58 I/O Hub GPIO and Scratch Pad
↪  Registers
/0/10d/14.2                     generic
↪  7500/5520/5500/X58 I/O Hub Control Status and RAS
↪  Registers
/0/10e                          bridge     Intel
↪  Corporation
/0/10f                          bridge     Intel
↪  Corporation
/0/110                          bridge     Intel
↪  Corporation
/0/111                          bridge
↪  7500/5520/5500/X58 Physical Layer Port 0
/0/112                          bridge
↪  7500/5520/5500 Physical Layer Port 1
/0/113                          bridge     Intel
↪  Corporation
/0/114                          bridge     Intel
↪  Corporation
/0/115                          bridge     Intel
↪  Corporation
/0/116                          bridge     Intel
↪  Corporation
/0/117                          bridge     Intel
↪  Corporation
/0/118                          bridge     Intel
↪  Corporation
/0/119                          bridge     Xeon 5600
↪  Series QuickPath Architecture Generic Non-core
↪  Registers
/0/11a                          bridge     Xeon 5600
↪  Series QuickPath Architecture System Address
↪  Decoder
/0/11b                          bridge     Xeon 5600
↪  Series QPI Link 0
/0/11c                          bridge     Xeon 5600
↪  Series QPI Physical 0
/0/11d                          bridge     Xeon 5600
↪  Series Mirror Port Link 0
/0/11e                          bridge     Xeon 5600
↪  Series Mirror Port Link 1
/0/11f                          bridge     Xeon 5600
↪  Series QPI Link 1
/0/120                          bridge     Xeon 5600
↪  Series QPI Physical 1
/0/121                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Registers
/0/122                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Target
↪  Address Decoder
/0/123                          bridge     Xeon 5600
↪  Series Integrated Memory Controller RAS Registers
/0/124                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Test Registers
/0/125                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0
↪  Control
/0/126                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0
↪  Address
/0/127                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0 Rank
/0/128                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0
↪  Thermal Control
/0/129                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1
↪  Control
```

```
/0/12a                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1
↪  Address
/0/12b                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1 Rank
/0/12c                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1
↪  Thermal Control
/0/12d                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2
↪  Control
/0/12e                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2
↪  Address
/0/12f                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2 Rank
/0/130                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2
↪  Thermal Control
/0/131                          bridge     Xeon 5600
↪  Series QuickPath Architecture Generic Non-core
↪  Registers
/0/132                          bridge     Xeon 5600
↪  Series QuickPath Architecture System Address
↪  Decoder
/0/133                          bridge     Xeon 5600
↪  Series QPI Link 0
/0/134                          bridge     Xeon 5600
↪  Series QPI Physical 0
/0/135                          bridge     Xeon 5600
↪  Series Mirror Port Link 0
/0/136                          bridge     Xeon 5600
↪  Series Mirror Port Link 1
/0/137                          bridge     Xeon 5600
↪  Series QPI Link 1
/0/138                          bridge     Xeon 5600
↪  Series QPI Physical 1
/0/139                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Registers
/0/13a                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Target
↪  Address Decoder
/0/13b                          bridge     Xeon 5600
↪  Series Integrated Memory Controller RAS Registers
/0/13c                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Test Registers
/0/13d                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0
↪  Control
/0/13e                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0
↪  Address
/0/13f                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0 Rank
/0/140                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 0
↪  Thermal Control
/0/141                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1
↪  Control
/0/142                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1
↪  Address
/0/143                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1 Rank
/0/144                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 1
↪  Thermal Control
/0/145                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2
↪  Control
/0/146                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2
↪  Address
/0/147                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2 Rank
/0/148                          bridge     Xeon 5600
↪  Series Integrated Memory Controller Channel 2
↪  Thermal Control
/0/d                 scsi0      storage
/0/d/0.0.0           /dev/sda   disk       60GB
↪  MK0060EAVDR
/0/e                 scsi1      storage
/0/e/0.0.0           /dev/sdb   disk       60GB
↪  MK0060EAVDR
/1                              power      578322-B21
/2                              power      578322-B21
/3                              power      578322-B21
/4                              power      578322-B21
/5                   ib2        network    interface
/6                   ib0        network    interface
```

ARTIFACT EVALUATION

Verification and validation studies: We verified the correctness of the routing and modified MPI library through network simulations, on a small test bed, and by checking if the used LIDs (via ibdump) and routing tables are as expected for the final large-scale supercomputer. For the benchmarks, we performed multiple runs, and investigated whenever abnormalities happened during benchmark execution, e.g., unexpectedly long executions were checked whether it was a persistent behavior or transient problem which can be overcome by rebooting a node or the system.

Accuracy and precision of timings: We primarily rely on the performance data provided by each application or benchmark (which we did not implement ourself). For instances, where we injected time measurement instructions to wrap the solver/kernel part of an (proxy-)application, we used standard MPI_Wtime calls and verified its correctness using a smaller test bed. The accuracy of MPI_Wtime is sufficient for our tests, which run for multiple minutes.

Used manufactured solutions or spectral properties: No new hardware was developed. All measurement- or performance-relevant parts of the newly wired HPC system belonged to the decommissioned system. All peripheral parts which we needed to retrofit the system, such as Ethernet cables for the management network or BIOS batteries to replace depleted ones, are of-the-shelf components.

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: We did not modify the system (hardware or software) during the experiment, except for rebooting crashed compute nodes or replacing faulty compute node, and had exclusive access. Whether replacing a broken node (with a similar spare node) altered the condition for the benchmarks is beyond our current evaluation capabilities. We pinned down, wherever possible, the software and hardware parameters such that (as far as we can tell) the only difference between benchmark executions is time and network topology (and routing/placement) as discussed in the paper.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. We had exclusive access to the entire system. MPI processes were allocated by host lists for repeatability, and OpenMP threads have been pinned to CPU cores. All executions, inputs, iterations, and the software environment, etc., were controlled by (bash) scripts. Other potential causes of run-to-run variability are outside our control, and hence we executed every application and benchmark 10 times for the exact same configuration of input, node allocation, scale (node count), topology, and routing. Consequentially, we report min/max/median and 1st/3rd quartiles for all application benchmarks and the absolute/relative best performance comparison for the pure MPI benchmarks.