J Supercomput (2016) 72:4418–4437
DOI 10.1007/s11227-016-1755-2

The BXI routing architecture for exascale supercomputer

Pierre Vignéras[1] · **Jean-Noël Quintin**[1]

Published online: 24 May 2016

Abstract BXI, Bull eXascale Interconnect, is the new interconnection network developed by Atos for high-performance computing. It has been designed to meet the requirements of exascale supercomputers. At such scale, faults have to be expected and dealt with transparently so that applications remain unaffected by them. BXI features various mechanisms for this purpose, one of which is based on a clear separation between two modes of routing tables computation: offline mode used during bring-up and online mode used to deal with link failures and recoveries. This new architecture is presented along with several offline and online routing algorithms and their actual performance: the full routing tables for a 64k-node fat-tree can be computed in a few minutes in offline mode; and the online mode can withstand numerous inter-router link failures without any noticeable impact on running applications.

Keywords Fabric management · Routing · Fault-tolerant routing · BXI · Interconnect management · High-performance computing

This BXI development has been undertaken under a cooperation between CEA and Atos. The goal of this cooperation is to co-design extreme computing solutions. Atos thanks CEA for all their inputs that were very valuable for this research.
This research was partly funded by a grant of Programme des Investissments d'Avenir.
BXI development was also part of PERFCLOUD, the French FSN (Fond pour la Société Numérique) cooperative project that associates academic and industrial partners to design and provide building blocks for new generations of HPC data-centers.

✉ Pierre Vignéras
pierre.vigneras@atos.net

Jean-Noël Quintin
jean-noel.quintin@atos.net

[1] Campus Ter@tec, 2 rue de la piquetterie, 91680 Bruyères-le-Châtel, France

1 Introduction

High-performance computing workloads must scale to extreme levels of parallelism with applications using tens of thousands of nodes and dozens or hundreds of threads on each node. BXI, Bull eXascale Interconnect, is the new interconnection network developed by Atos and designed to meet these requirements. The BXI interconnect is based on two ASICs: a network interface controller (NIC) and a 48-port switch. The BXI switch is a 48-port crossbar featuring per-port destination-based deterministic routing and adaptive routing. BXI technology also contains various features at hardware levels (NIC and switch) to ensure that faults are unnoticed by applications if routing tables are modified in a given time frame—10 s currently, including notification, computation and upload of modifications. This value is a trade-off between resiliency and actual failure detection latency: increasing this timeout might help improving resiliency for some failures, but can also lead a job to halt for too long before detecting there is no way to bypass a failure. A detailed overview of the BXI technology can be found in [1].

To use this infrastructure, routing tables must be initially computed and loaded into the different BXI switches through an out-of-band network. Each table defines how to reach an end-node (NID for Node IDentifier) based on its destination address and the switch input port. The computation of these routing tables is not trivial, since, regardless of topology, it must guarantee the absence of:

– **Deadlock** messages can no longer move in the network due to lack of available resources;
– **Livelock** messages are constantly moving in the network but never reach their final destination—as within a loop;
– **Dead-end** messages reach an unusable port (nonexistent, not connected, or down).

The routing tables must be quickly computed in order to keep the system operational without interrupting running applications. While a few seconds delay is acceptable, standard algorithms require dozens of minutes when applied to exascale system topologies with tens of thousands of nodes. Given the size of these systems, component failures[1] might be expected on a regular basis. The naive approach where all routing tables are re-computed from scratch for each failure is not acceptable at such scale. The two main reasons are:

– the computation time exceeds a timeout: faults are therefore not hidden to applications (*soft real-time requirement*);
– the modification of all routing tables impacts all running applications even when only a small part of the platform is concerned (*minimum impact requirement*).

This document presents our routing strategy for BXI exascale supercomputers. The state of the art is first reviewed in Sect. 2. The major problems we address are then detailed in Sect. 3. Our solution is described in Sect. 4 along with some results. Finally, we conclude in Sect. 5 and explain the directions for future developments.

[1] The term “failures” must be understood from a fabric management point of view. Hardware failures are of course seen as such, but also human mistakes and maintenance operations.

2 Related works

There is an abundant literature on supercomputing interconnect topologies and their related routing algorithms [2,3]. In the recent past, topologies designed for high-radix switches have been proposed: fat-trees [4–7], flattened-butterfly [8,9], dragonfly [10,11] or slimfly [12], are such examples. However, to our knowledge fault management is rarely addressed in the literature for topology-specific routing algorithms. Most of the time, a link or switch fault is considered as a whole topology change, triggering routing tables re-computation from scratch and requiring their complete upload. This option is not practical anymore for exascale supercomputers mostly because of computational complexities in time as explained in Sect. 3. When the topology-specific routing algorithm cannot be used anymore on the degraded topology, the computation switches to a topology-agnostic routing algorithm [13–16]. In such a case, the routing tables are also computed from scratch and uploaded; such an operation is far too heavy to meet our soft real-time and low impact requirements. Degraded topologies must therefore be supported as far as possible for a given routing algorithm.

The shortest path computation is the basis of most topology-agnostic routing algorithms [17]. Recent results in the dynamic solving of All-Pairs Shortest Paths Problem (APSP) [18–20], can help dealing with failures: a slight modification of the topology can be performed in $O(f(n) + n^2 log\, n)$ (for most dynamic APSP solutions), and the re-computation of routing tables can use the fast query time in $O(1)$ (for all dynamic APSP solutions). However, for exascale topologies, the corresponding graph and memory consumption associated with dynamic APSP solutions makes their application unusable in practice.

Fault-tolerant topology-agnostic routing algorithms have been proposed [21–24]. Such solutions do not meet our requirements, since they support only one fault or are too slow for exascale topologies. Moreover, they often focus on the support of irregular topologies, producing routing tables that may be of lower quality than specific routing algorithms.

Recently, the Quasi-Fat-Tree topology has been proposed [25] along with a fault-tolerant routing algorithm in closed-form allowing for an efficient parallel implementation. This is an elegant solution to the problem but only available for quasi-fat-tree topologies, whereas our solution proposes an efficient general architecture suitable to any topology.

3 Major issues

Computing routing tables for a topology containing N destinations (NID) requires at least $\Omega(N^2)$ steps. Indeed, for each destination, a route must be found to reach any of the other $N - 1$ destinations. That means, for targeted exascale topologies, with 64k NIDs, a minimum of 4 billion routes must be computed. Soft real-time requirement imposes a 5-s constraint to routing tables computation giving 5 other seconds for notification and upload. Therefore, a rate of 860M routes per second must be achieved.

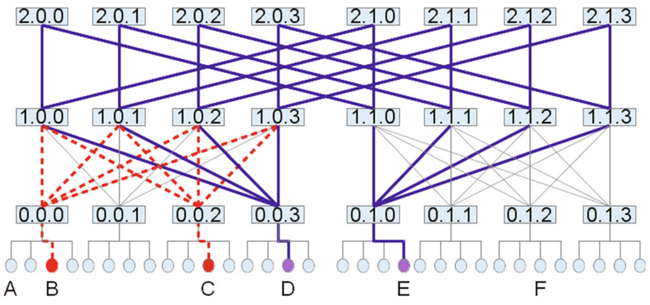

Fig. 1 Number of routes in a fat-trees with adaptive routing. Switch labeling for this example will be described in Sect. 4.1.2

For comparison purposes, on Curie,[2] a fat-tree with 5739 nodes, the Infiniband subnet manager (opensm) computes the routing tables in 14 s achieving a rate of 2.3M of routes/s. Another similar result is given in [25]: the `ftree` opensm routing algorithm computed the routing tables of a fat-tree with 34,992 nodes in 478 s giving a rate of 2.5M routes/s. This is 400 times slower than our objective.

Furthermore, with BXI-adaptive routing feature, a much larger number of routes must be computed as shown by Fig. 1 representing a fat-tree. The number of routes to compute depends on the source–destination pair; it increases significantly with the nodes distance: there is a single route from A to B through switch 0.0.0, 4 routes (red dashed-lines) from B to C and 8 (purple thick-lines) from D to E. The total number of computed routes is therefore significantly higher than N^2. On a l-level full fat-tree made of radix-k switches, with $k/2$ uplinks at each level i, $i < l$, the total number of routes for a given pair of source–destination can be as high as $(k/2)^l$. In the case of a 64 k nodes 4-level BXI fat-tree, for a single source–destination pair with a maximum distance, that means $(48/2)^4 > 300k$ routes to consider.

From another perspective, if S is the total number of switches in the topology, the routing component must:

– assign integers between 0 and 47 (each switch has 48 ports in BXI) to all $S \cdot 48 \cdot N$ entries of deterministic routing tables (which are 1 byte each);
– set each 48 bits (6 bytes) of $48 \cdot S \cdot (N/8) = 6SN$ entries of adaptive tables (1 entry for a group of 8 consecutive NIDs, hence $N/8$ entries per switch port).

Consequently, for a BXI topology with $N = 62{,}208$ NID and $S = 9504$ switches, the routing algorithm must fulfill 28.3G deterministic entries and 3.5G adaptive entries for a total of about $84 \cdot S \cdot N = 49\,G$ bytes.

Keeping our 5-s constraint (soft real-time requirement), an ideal processor clocked at 3 GHz must compute $49\,\text{GB}/(5\,\text{s} \cdot 3\,\text{GHz}) = 3.3$ bytes per clock cycle. This performance is close to the limit achievable with modern processors in various benchmarks,[3] especially with non-vectorized instructions. This basically means a brute-force generic

[2] Curie is an instance of the previous petaflopic generation of Bull supercomputer. It was ranked 9th in the Top500 list of June 2012: http://www.top500.org/system/177818.

[3] https://gmplib.org/~tege/x86-timing.

solution to the initial problem of real-time routing tables computation is not a good approach anymore.

4 Solution

The main idea of our proposal relies on a clear separation between two distinct modes of operation: offline mode and online mode. In order to evaluate our solution, a Bull R428 server with 4 Intel Xeon CPUs E5-4640 (8 cores) clocked at 2.40 GHz, 512 GB of RAM and a single disk was used.

4.1 Offline mode

The offline mode is used mainly during bring-up phases (installation, extension, maintenance) but also for testing purposes and offline analysis. Routing tables are computed from a nominal topology, where all equipments are considered up and running (even equipments expected to be installed afterwards). Those routing tables are usually uploaded after a validation phase: checking that neither deadlock, livelock nor dead-end are induced by these routing tables. Moreover, a quality check can also be performed according to some criteria such as load balancing, communication patterns, and so on. Since it is performed offline, this computation has no impact on any part of the running supercomputer until the output is uploaded, and there is no stringent limit on the time required for the computation.

A given offline routing algorithm is a trade-off between genericity—it provides valid routing tables for any topology—and performance—the computed routing tables are of good overall quality. Implementing an offline routing algorithm is easy to do thanks to a well-defined API. BXI already implements several offline routing algorithms shortly described below.

4.1.1 Topology-agnostic offline routing algorithms

A routing algorithm is topology-agnostic if it does not depend on the actual topology layout or addressing scheme for its routing decision. Those algorithms target two kinds of topologies:

- graph-based topologies with no specific layout;
- degraded specific topologies based on well-defined topology classes such as torus, mesh, hypercube, pgft, hyperx, dragonfly among others, but with some differences that make them unsuitable for their related specific routing algorithms (either because the routing algorithm cannot route those degraded topologies, or for performance reasons).

Note however, that some topology-agnostic routing algorithms might not be deadlock-free meaning that for some topologies, their computed routes can lead to the deadlock of some fabric part where no more messages can be neither sent nor received from some switches.

Topology-agnostic offline routing algorithms currently implemented in BXI are shortly described below.

MinHop is based on the Dijkstra algorithm [26] to find shortest paths between all pairs of source/destination, it selects one random minimal route as the deterministic route for such a pair, and all minimal routes for their adaptive routes. It is topology-agnostic but not deadlock-free. It can therefore be used on any topology where minimal routing does not lead to a deadlock (see [27]). This is the case for most topologies seen as pseudo-fat-trees but that cannot be expressed as PGFTs.

UP/DOWN** [28] is given a vertex in the graph called the root, from which a spanning tree is built and restrictions are inserted in such a way that all messages from any given leaf must follow a *valid path*: going up towards the root before going down towards its destination leaf. However, it is not required to reach the root in order to turn down: as soon as a valid path is available to reach the final destination it can be used.

Since a spanning tree has been built, all leaves can be reached unless the graph itself is not connected. Since messages always move up then down and never the reverse the algorithm is deadlock-free and can be used with any topology.

The spanning tree is used to define restrictions: each edge of the topology graph is marked as either uplink or downlink. Note that an uplink for a given vertex is a downlink for the vertex at the other end. Uplinks are preferable because they allow more path diversity: a downlink can follow an uplink. The spanning-tree is therefore computed with a breadth-first search in order to increase the number of uplinks leveraging the adaptive feature of BXI.

By default the current implementation uses only minimal valid paths. As a side effect, when applied on a fat-tree with a leaf as the root of the spanning tree, all valid paths are "natural" routes of the fat-tree. Therefore, the computed routing tables becomes statistically equivalent to both `minhop` and `pr1tp`. As counter intuitive as it seems to be, this root selection gives the best result when applying the `up*/down*` algorithm on a fat-tree. The actual formal reasons behind this statement is beyond the scope of this paper.

4.1.2 Topology-specific offline routing algorithms

A routing algorithm is topology-specific if it depends on the actual topology layout and/or addressing scheme for its routing decision. All topology-specific routing algorithms are deadlock-free when applied to their targeted topology. Of course a topology-specific routing algorithm usually does not work on a topology that is not in its targeted class. For example, applying a PGFT-specific routing algorithm on a Torus topology will not work in three distinct ways:

- the routing algorithm will detect it does not support the given topology and will refuse the production of routing tables;
- the routing algorithm is forced to continue anyway, and the topology does not contain the required data (specific information about the topology such as switch

upports in a fat-tree). In this case the routing algorithm will refuse the production of routing tables;

– the routing algorithm is forced to continue anyway and the required data are available. In this case, some or all routing tables might well be produced but the behavior of the fabric after those routing tables have been uploaded to their related switches is undefined.

Topology-specific offline routing algorithms currently implemented in BXI are shortly described below.

Pr1ts (PGFT Random 1 Table per Switch) is designed for a wide set of fat-trees (any k-ary n-tree [6], XGFT [5] or PGFT [7] actually). It is based on `random` [6,29]: it basically selects one switch randomly among the set of Nearest Common Ancestor (NCA) for a given source–destination pair. In a fat-tree, only two directions are possible for a given message: up or down. `pr1ts` relies on the specific addressing scheme in order to decide:

– each switch is given a topological address which is a tuple (level, $d_l, d_{l-1}, \ldots, d_1, d_0$);
– a switch at level l is connected to all switches at level $l + 1$ that have all their digits in common excepting digit at level l that can be different. For example, in Fig. 1, switch (1, 1, 3) is connected to all switches at level 2 that has the following address: (2, $*$, 3). Note that it is also connected downward to all switches with the following address (0, 1, $*$).

Therefore, determining if a message must go up or down from a given switch is done thanks to a comparison between the current switch and the destination leaf: a message must go down when the current topological address and the destination address share the same prefix up to their level difference. After this comparison, 1 port is selected randomly among all the available ports for the deterministic route. All available ports are selected for adaptive routes. `pr1ts` computes only 1 table per switch, and assign the same routing table to all ports of the same switch. This leads to very fast computation. On some communication patterns, it can provide better performance than `d-mod-k` or `s-mod-k` [30].

Pr1tp (PGFT Random 1 Table per Port) is similar to `pr1ts`, it computes however one routing table per port leading deterministic routes to be more spread out over the whole topology than with `pr1ts`. According to the communication pattern it can offer better or worse performance that `pr1ts`. It can also be used on a wide set of fat-trees.

D-mod-K [30] is designed for Real Life Fat Trees [7]—a subset of PGFTs with several constraints[4]—and guarantees non-blocking traffic for shift permutations on them. As with `pr1ts` and `pr1tp`, the algorithm is entirely based on the addressing scheme. For each switch, it applies a simple formula to decide how to reach a given destination address. This formula spreads the traffic deterministically over the whole topology in

[4] In particular, the number of connected ports of each switch must be the same, top switches included.

a perfect load balanced way. It also ensures that contention due to the reaching of a same destination from different sources happens only during the upgoing path, never on the downgoing path. Note that `d-mod-k` computes 1 routing table per switch, and assign it to all ports of the same switch. This is in contrast to `s-mod-k`.

S-mod-K [5,30] is similar to `d-mod-k` but the formula is based on the source address instead of the destination address. Therefore, it distributes sources to top-level switches of a fat-tree. Thus, contention due to the reaching of a same destination from different sources happens during the downgoing path, never on the upgoing path. Since routing tables are destination based, the implementation of this algorithm requires 1 table per port. Depending on the communication pattern, it can offer better or worse performance than `d-mod-k`.

4.1.3 Offline mode computation time experimental results

Even if the offline mode is not time-bounded from a functional perspective, it remains softly time-bounded from a user perspective. According to time and space complexities discussed in Sect. 3, a naive implementation leads to unreasonable execution time (dozens of minutes). This problem is also described in [7,25] and proposed solutions consist in using a closed-form routing tables computation, allowing for an efficient parallel implementation but limited to few topology/routing algorithm combinations.

Offline routing algorithms for BXI are implemented using a big set of purely software optimization techniques: lock-less multi-threaded design, thread binding, NUMA memory binding. Profiling shows the process is largely I/O bounded: the design has been changed to reduce drastically the number of system calls—using mmap() and an asynchronous logging library for example. These improvements make the offline mode scalable up to the maximum of available cores.

Results are presented in Fig. 2 for a fat-tree containing 57,600 NIDs, 9600 switches and 153,600 inter-switch links. The average time of 60 executions is given. In the best case, 4 s only are required to compute the routing tables producing $84 \cdot S \cdot$

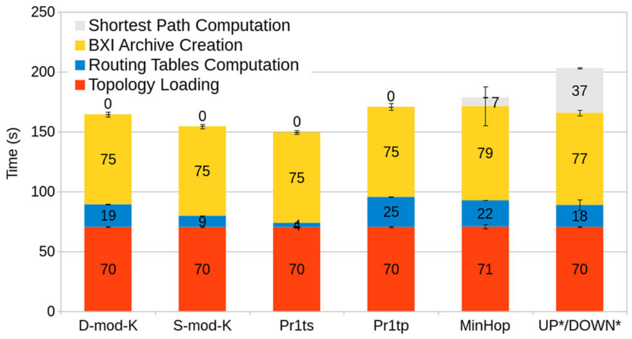

Fig. 2 Execution time in seconds of various BXI offline routing algorithms

$N = 46G$ bytes. Therefore rates of 829M routes/s[5] and 4.8 bytes per clock cycle are achieved. This results from the speedup of our parallel implementation of offline routing algorithms. However, as shown in the figure, most of the time is taken by input/output operations: topology loading (in red) and BXI archive creation (in green). Considering all steps gives 22M routes/s and 0.12 bytes per clock cycle for the best case. Though ten times faster than opensm, this might seem low compared to objectives given in Sect. 3. Nevertheless, since offline, only the end-user perspective must be taken into consideration: around 3 min are required in all cases to compute and to store the whole set of deterministic and adaptive routing tables for such big topologies on a single disk. This is considered reasonable.

4.1.4 Offline mode routing algorithms performance comparisons

The performance of a given routing algorithm depends on several factors such as topology, communication pattern, message size, adaptive routing, among others. In the following experiment, a 3-level full fat-tree formally defined by

$$\mathrm{PGFT}(3; 24, 12, 4; 1, 12, 4; 1, 2, 2)$$

holding 1152 nodes is used with the N-pair communication pattern which is a subset of shift all-to-all described in [31,32]. This communication pattern has been selected in our initial study because it is found in several important collective communication implementations. N-pair is defined by the following scheme: the topology is cut in two halves such that communication occurs between a pair of nodes of each half. Formally, nodes are indexed from 0 to $N - 1$ and node i communicate with node $N/2 + i$ with $0 \leq i < N/2$ and vice-versa. Traffic therefore always reaches top switches for this communication pattern preventing congestion-less bias due to local-only traffic such as with a pattern where each node communicates with its immediate neighbor. The simulator used in this experiment is an home-made product called CoSIN (Composition and Simulation of Network), that models the BXI switches at transaction level in SystemC. At the end of a simulation session, CoSIN produces various metrics including the total simulation time. As Transaction Level Modeling adopts an approximated time, the time unit is expressed as an abstract "cycle" time. This metric is relevant to compare overall performance of different routing algorithms.

For a given offline routing algorithms, routing tables are computed, injected into CoSIN which then starts the simulation of the N-pair communication pattern. This is reproduced 15 times for each routing algorithm. The average total simulation time in TLM abstract cycle is then used to compare the performance of routing algorithms.

Figure 3 presents results when 80 % of message are 120 bytes, all 20 % others are 64k bytes. As expected, `d-mod-k` and `s-mod-k` which are specifically designed for such a pattern on such a topology provide the best performance. Notice that their performance are almost identical (0.02 % difference). This conforms to the result given in [30] where both algorithms are proven to be equivalent on such a pattern.

[5] The actual rate is much higher since only deterministic routes are considered in this rate computation.

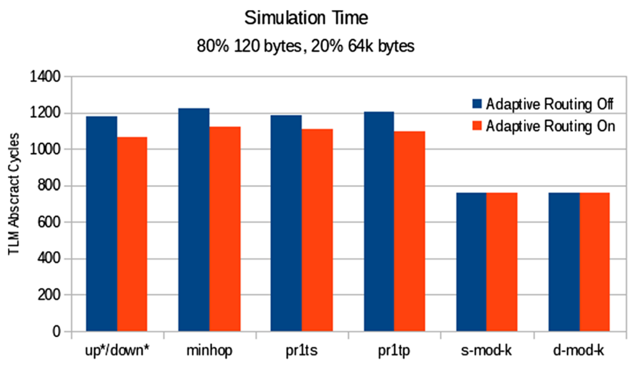

Fig. 3 Simulation time for the N-pair communication pattern with 80 % of 120 bytes and 20 % of 64k messages on a 1152 3-level fat-tree

Moreover, there is no significant difference between all others. The reason holds in the way routes are spread out. Except `d-mod-k` and `s-mod-k`, all others use a random selection among valid routes. For `minhop`, valid routes are all minimal routes which are "natural" routes in a fat-tree (up towards the nearest common ancestor, then down). For `up*/down*`, as explained in Sect. 4.1.1, selecting a leaf switch as the root of the spanning tree leads all valid paths to be "natural" routes of the fat-tree. For `pr1ts` and `pr1tp`, valid routes are—by design—"natural" routes in a fat-tree. Therefore, routing tables computed by `minhop`, `up*/down*`, `pr1ts` and `pr1tp` are statistically equivalent.

Adaptive routing improves the overall performance for most routing algorithms on this communication pattern (between 6.5 and 9.6 %). The reason why it does not have any impact on neither `s-mod-k` nor `d-mod-k` holds in the absence of congestion. The impact of adaptive routing in fat-tree-specific routing algorithms has already been discussed [33,34]. In BXI, a message is candidate for adaptive routing only if in-order guarantee is not required and if its size is above a customizable threshold (320 bytes).

Figure 4 presents the result when 100 % of messages are 321 bytes, that is one byte more than the threshold above which messages are candidate for adaptive routing. Again, `d-mod-k` and `s-mod-k` exhibit the best result with no significant difference. However, adaptive routing improves performance by a significant factor for `minhop` (29 %), `up*/down*` (28 %), `pr1ts` (32 %) and `pr1tp` (29 %). The performance improvement for `d-mod-k` and `s-mod-k` is not significant because of the absence of congestion.

4.2 Online mode

The online mode is used while the system is up and running. This mode deals with faults and recoveries, computing small routing tables modifications while guaranteeing the absence of deadlock, livelock and dead-end. Since it is online, the computation of the routing table modifications must be completed in less than 5 s in order to limit the impact of the fault on the jobs using the faulty link. Note that the BXI switch

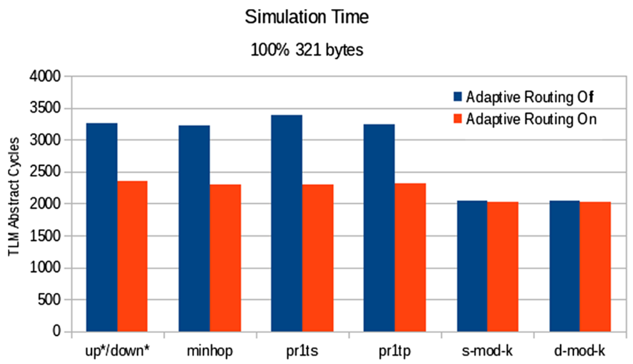

Fig. 4 Simulation time for the N-pair communication pattern with 100 % of 321 bytes messages on a 1152 3-level fat-tree

adaptive routing feature is targeted towards improving communication performance as shown in [34,35], not fault management. The set of routing table modifications must also be kept to a minimum to limit the impact on jobs not using the faulty link. Moreover, any node must be able to communicate with any other node in the topology unless the topology graph is disconnected. Note that this last property is not possible for all online routing algorithms: a highly degraded fat-tree for example, might be routable by agnostic routing algorithms only such as `minhop`. Consider for example the case where on the fat-tree shown in Fig. 1, uplinks 0.0.0–1.0.x with $x \in [1, 3]$ and uplinks 0.0.3–1.0.y with $y \in [0, 2]$ are all faulty. Minimal routes from B to D are given by the following path 0.0.0; {2.0.0, 2.1.0}; 1.1.0; 0.1.0; 1.1.3; {2.0.3, 2.1.3}; 1.0.3; 0.0.3. Such paths cannot be computed by a fat-tree-specific routing algorithm, but are provided by minhop. Note however that minhop is not deadlock-free whereas any fat-tree-specific routing algorithm is by design. Therefore, any online routing algorithm raises an error when it detects such a situation (as discussed in the introduction, switching to a new routing algorithm is not an option).

4.2.1 Architecture

In online mode, the `Routing` component must react to events sent by any fabric equipment. For this purpose, the `Backbone` component acts as a middleware bus: it receives topology updates from other components and publishes them to the `Routing` component shown in Fig. 5.

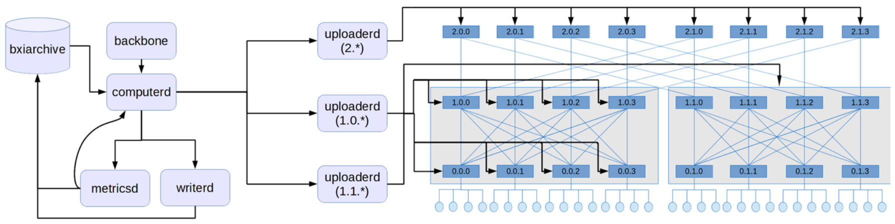

Fig. 5 The `Routing` component architecture

The `Routing` component is actually made of different parts which can be distributed between several management hosts:

- `Computerd`: it starts from a *bxiarchive* that includes routing tables previously written to a storage device by the *offline* mode. It then receives updates from `Backbone` and computes the required modifications of routing tables called `rtmods` before publishing them to subscribers.
- `Writerd`: it also starts from a *bxiarchive* and subscribes to all routing tables modifications published by `Computerd.` It applies all received `rtmods` to its own memory model of the actual topology and writes down the result to the storage device, making a new *bxiarchive* (only new data are created, unmodified data are hard linked to reduce storage space). Note that in BXI, an archive is produced each time modifications are taken into consideration by the `BXI Fabric Management`. The set of archives allows for various analyses such as post-mortem debugging, study of the timeline of events, fault handling optimizations, …
- `Metricsd`: it also starts from a *bxiarchive* and also subscribes to all routing tables modifications published by `Computerd`. It also applies all `rtmods` received to its own memory model of the actual topology. It then computes various *metrics* on the result (such as link load balancing, number of sources and destinations for each route traversing each port, …), and provides its feedbacks to `Computerd`. This may help the online routing algorithm adapt its internal parameters to provide better `rtmods` according to some criteria measured my `Metricsd`. This is an ongoing work not presented in this paper.
- `Uploaderd`: its role is to transform `rtmods` format into uploadable routing tables modifications format, and to upload them to the concerned switches using the SNMP protocol. Traffic is not stopped during the upload. A short discussion on deadlock-free dynamic reconfiguration is given in Sect. 4.2.3. In the new Bull eXascale platform, a "cell" is composed of three cabinets with up to 288 nodes. In such a cell, all switches are not directly available through out-of-band SNMP protocol for various reasons.[6] Therefore, a given `Uploaderd` acts as a proxy for the set of switches it manages, given their shared topological address prefix. Using this prefix, it subscribes to a subset of `Backbone` updates. Thus, an `Uploaderd` receives only `rtmods` related to the switches it is concerned with, thus limiting the amount of data received and managed.

As an example, using the notation defined in [7], the fat-tree formally described by PGFT(4; 24, 12, 15, 15; 1, 12, 24, 5; 1, 2, 1, 3) contains:

- 64,800 NIDs, 11,160 switches and 194,400 inter-switch links;[7]
- $12 \times 15 \times 15 = 2700$ switches at level 1, each connected to 24 nodes and 12 L2 switches with two links;

[6] For any two cells, switches at same relative location are assigned same private non-routed IP address for out-of-band communication. This ease delivery, bring-up and also optimize the out-of-band management network. More details in [1].

[7] Only inter-switch link faults can be dealt with by an online routing algorithm.

- $12 \times 15 \times 15 = 2700$ switches at level 2, each connected to 12 L1 switches and 24 L3 switches with one link;
- $12 \times 24 \times 15 = 4320$ switches at level 3, each connected to 15 L2 switches and 5 L4 switches with three links;
- $12 \times 24 \times 5 = 1440$ switches at level 4, each connected to 15 L3 switches.

For such a topology, the set of `Uploaderd` is defined by the following schema:

- $15 \times 15 = 225$ instances for the management of groups of 24 L1/L2 switches;
- $24 \times 15 = 360$ instances for the management of groups of 12 L3 switches;
- 5 instances for the management of groups of 288 L4 switches.

Thanks to the high-performance zeromq[8] socket library, this architecture scales up to several hundreds of `Uploaderd`: $225 + 360 + 5 = 590$ in this extreme case. For other topologies, the layout of `Uploaderd` follows the same principle of hierarchical layout.

Note that the online architecture follows a pipeline design: `Computerd` always receives a batch of topology updates consisting of link failures and link recoveries. A specific message received from the `Backbone` triggers the actual computation on the whole set of updates. During that time, new updates are buffered. As soon as `Computerd` has published the computed `rtmods`, it starts working on next buffered updates. This way, simultaneous failures and failure happening during failure management are dealt with in a consistent manner.

BXI features a separate management network that is used for communication between all routing components such as `Backbone`, `Computerd`, `Writerd`, `Uploaderd`, `Metricsd` and all switches. Therefore, the time requires to react to fabric events does not depend on the fabric status itself (congestion, failures, and so on).

4.2.2 Online routing algorithms

An online routing algorithm is run by `Computerd` to react on minor modifications in the fabric such as failure and recovery of ports, links and switches. It updates routing tables so each pair of NIDs is given at least one valid route. Those tables updates are immediately uploaded to their related switches so the whole fabric recovers its state as fast as possible. These new tables are also dumped to a new bxiarchive by `Writerd`.

Note that *minor modifications* **are not** considered *topology changes*: the in-memory model of the topology does not see any objects addition or removal. This contrast with *major modifications* such as port, link or switch addition (nodes apart) that is not dealt with by online routing algorithms. For such cases, the topology **must** be changed and validated before hand using a specific well-defined workflow.

A formal description of the two BXI online routing algorithms described shortly below is given in [36].

ftrnd_diff is based on the fat-tree addressing scheme to bypass a failed port and re-balance routes on port recovery. It defines the concept of *twin switches*: two switches

[8] http://zeromq.org/.

are *twins* if they are connected to the same set of upper-level switches. This property is formally defined by the fat-tree addressing scheme—twins share the same address suffix of length the switch level in the fat-tree. As an example, in Fig. 5, switch 1.0.2 and 1.1.2 are twins as they share the same suffix ‘2’. Note that there are no twin at first and last levels.

When a link fails between any two switch S_a and S_b, an alternative link between S_a and S_b is looked for. On some fat-trees (e.g., PGFTs), such an alternative exists and the failure is dealt with in a straightforward manner. However, if such an alternative is not found, two cases must be distinguished:

– the failed port is an upport: any other upport leading to a usable twin can be selected; if such an upport does not exist because all upports are either faulty or lead to a faulty twin, the whole switch must be bypassed in order to reach some leaves (according to the current routing tables). Bypassing a switch means modifying the routing tables of all switches below it.
– The failed port is a downport: there is no alternative and since in a fat-tree, there is only a single downpath (from switch to switch), the whole switch must be bypassed. The only way to bypass a switch from the upside, is to bypass all its twins, that is to find uppaths that do not cross twins for all impacted leaves.

When a port is recovered, a check is first made to determine for each leaf l in the topology, if it was already reachable before the recovery. In the positive case, a simple rebalancing of routes is performed on the switch s owning the recovered port. In the negative case, l becomes reachable after this recovery, the related switch s can be used again to reach it and two cases must be distinguished:

– the recovered port is an upport: switches below s are threaded recursively in order to allow the reaching of l through s;
– the recovered port is a downport: routing tables on all twins of s must be modified in order to allow the reaching of l through s.

bsta_diff is based on the `up*/down*` offline topology-agnostic deadlock-free algorithm. It keeps this same property while remaining online. It recomputes deadlock-free modifications of existing routing tables in order to either bypass port failures or to rebalance routes on port recoveries. Basically, when a port fails, the shortest path towards all leaves is recomputed from the switch which handles the failure. The algorithm works in three steps:

– *Propagation* since a port has failed, its distance towards several leaves is set to infinity, and this is propagated towards the whole topology;
– *Recomputation* for each leaf in the topology, its shortest paths is recomputed taking restrictions into consideration;
– *Selection* new routes are selected according to the new shortest paths, and routing tables updated accordingly.

Note that the root switch selected for the spanning-tree computation and used for the setup of restrictions, is not considered as a special case by `bsta_diff`. If the root switch fails, all restrictions remain as they were initially computed, only distances are updated.

4.2.3 Transient fabric state

While uploading new routing tables, the fabric is for a limited time into a transient state where some switches have been updated while others will soon. In the general case, this might lead to deadlock [13].

In the specific case of fat-trees, fat-tree-specific offline and online routing algorithms are deadlock-free by design: a message always move up towards the NCA and then moves down towards its final destination.

In the case of our `bsta_diff` topology-agnostic online routing algorithm, routing tables modifications never change restrictions previously computed and set-up in the whole topology (cf. Sect. 4.1.1). Therefore, the same applies: messages in the fabric always move up the logical tree and then down, thus never reach—by design—a deadlock situation.

Both online routing algorithms are *transitively deadlock-free*: the routing tables updates computed and uploaded by our online routing algorithms are guaranteed to not introduce any deadlock as long as the original routing tables—computed by an offline routing algorithm—is itself deadlock-free. This property is formally defined in [36].

4.2.4 Online mode experimental results

To evaluate the online architecture presented in Sect. 4.2.1, the following experiment has been designed and evaluated:

– a random link fault is generated, and the related trigger is sent to the `Backbone`;
– `Backbone` publishes the trigger to all subscribers (only `Computerd` in our case);
– `Computerd` handles the fault with the computation of routing tables modifications (`rtmods`);
– `Computerd` then publishes all `rtmods` to subscribers;
– `Writerd` writes down received `rtmods` into a new *bxiarchive*.

The topology used is the 64,800-node fat-tree defined in the previous section. Note that it has two links between two L1/L2 connected switches, only one interlink between L2/L3 connected switches and three interlinks between L3/L4 switches. The impact of the number of links connecting a pair of switches on the computation of `rtmods` is discussed below.

For our experiment, first, we simulated 48k random link failures representing 25 % of the total number of inter-switch links, which is much more than expected in production use. Even with such a degraded fabric, the system is still able to route all the traffic correctly. The 48k missing links are then recovered and re-inserted in the fabric.

The offline and online routing algorithms are `pr1tp` (cf. Sect. 4.1.2), and `ftrnd_diff` (cf. Sect. 4.2.2), respectively. Of course, the presented results depend on the actual pair of offline/online routing algorithms chosen. However, the results presented here provide enough information to confirm the interest of the online architecture we propose. Other experiments with different combinations of offline/online routing algorithms are under development.

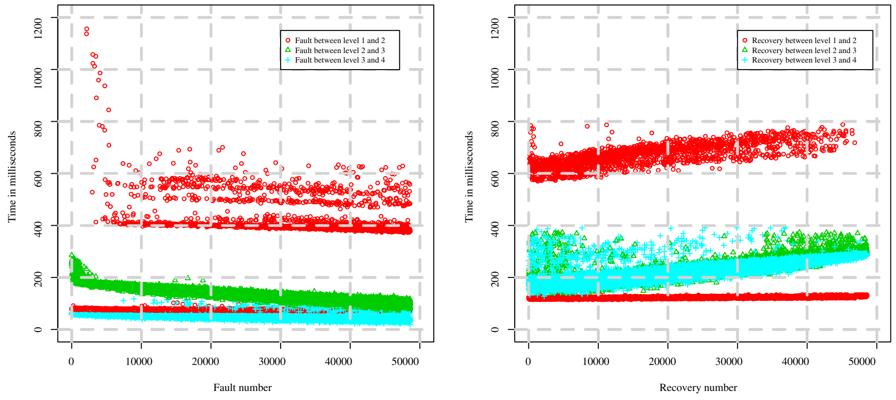

Fig. 6 Processing time for each fault (*left*) and recovery (*right*). *x*-axis represents somewhat a discrete timeline

Figure 6 presents the processing time for each fault and recovery: that is the time between the reception of the trigger by `Computerd` and the reception of all `rtmods` by `Writerd`. The processing time therefore includes both the computation time of `Computerd` and the sending time between `Computerd` and `Writerd`. The different points are colored according to the level of the failing/recovering link.

Globally, there is a tendency towards a reduction in the processing time for fault handling. As faults occur, there are fewer possibilities for alternative routes, reducing the number of `rtmods` to process. As expected, the exact opposite behavior is shown for recoveries: the processing time increases with the number of newly available paths.

L3/L4 faults are the easiest to deal with in our example, and they exhibit the best processing time. Note that several points for L3/L4 faults cost more than 100 ms (starting at around fault#8000), whereas most are closer to 60 ms. This is due to the number of interlinks between L3 and L4 switches: since three links connect them, when one link fails, dispatching the routes to the other two is fast and straightforward and impacts as few as 2 switches only, thus limiting the number of `rtmods` to compute. However, when no usable link remains between these two, many other switches must be impacted and the number of `rtmods` is much higher. This is also the reason why an L1/L2 link fault can cost around 80 ms in some cases and close to 1180 ms in others. Since there are 2 links between L1 and L2 switches instead of 3 between L3 and L4 ones, the worst case happens more often. Moreover, when there is no usable link between two L1/L2 switches, the impacted switches are all 2700 L1 switches. This case reaches the maximum number of impacted switches by the algorithm. Since the chosen topology provides a single link between L2 and L3 switches, this is always the most complex case: more than two switches are always impacted.

In Fig. 7, right-most values represent the maximum number of `rtmods` produced, which happens when first L1/L2 faults are encountered. Afterwards, if some links are already unusable, there is no `rtmod` to compute for their related ports. The same applies to (137k `rtmods`, ~60 ms) which is the maximum number of L3/L4 faults and also corresponds to first times such faults are seen.

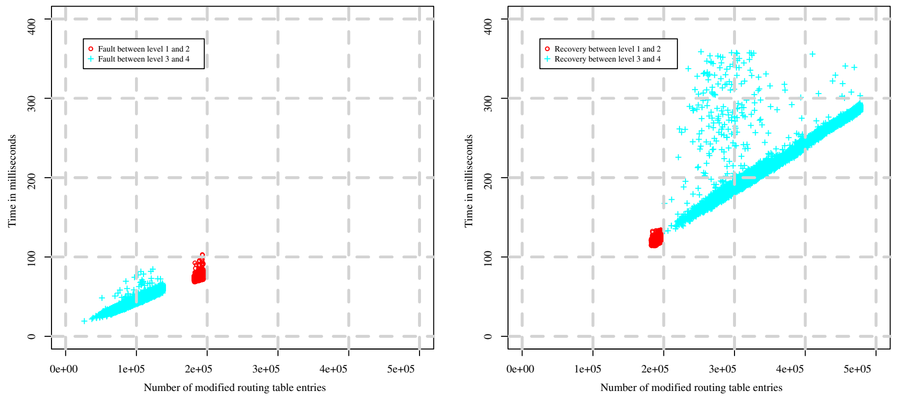

Fig. 7 Processing time for a given amount of `rtmods` produced when only two switches are impacted on fault (*left*) and recovery (*right*). L2/L3 link failure involves more than two switches, hence they do not appear in *figure*

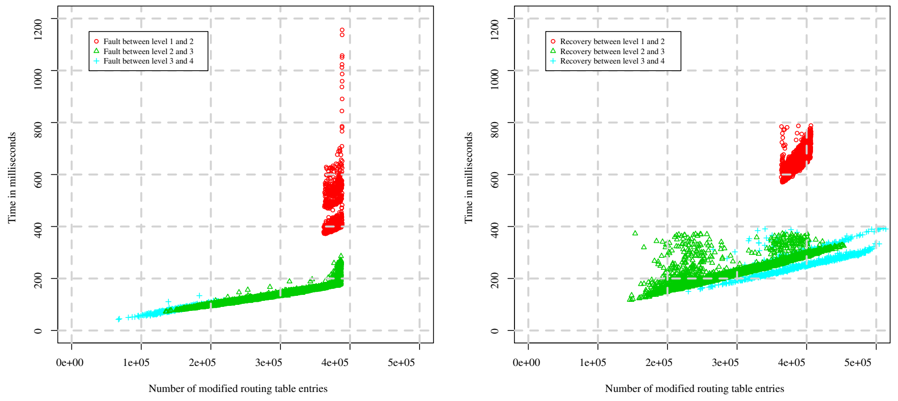

Fig. 8 Processing time for a given amount of `rtmods` produced when more than two switches are impacted for fault (*left*) and recovery (*right*)

Figure 8 presents the processing time for a given number of `rtmods` when more than two switches are impacted. The processing time depends on the level: it can take up to 1200 ms to deal with one worst-case L1/L2 fault while it takes only 140 ms to deal with an L3/L4 one.

Handling recoveries should exhibit a similar behavior: the number of `rtmods` is expected to be roughly the same on fault handling than on recoveries. This is not the case as shown in Fig. 7 where 15,000 `rtmods` maximum is computed for L3/L4 fault handling but at least 21,000 `rtmods` for recovery handling. Our algorithm computed more `rtmods` for recoveries than required. The problem has been identified and the correction is under development.

The BXI offline routing tables checker has been used at 10, 20 and 25 % of faults, and 10, 20 and 25 % of recoveries in order to check the validity of the computed routing tables ensuring the absence of deadlock, livelock and dead-end.

Checking the overall quality of the new routing tables is a challenge we are currently working on. Studying other topologies than fat-tree—thanks to BXI topology-agnostic online routing algorithms—is also an ongoing study.

5 Conclusion and future works

For the exascale topology size targeted by BXI technology, the complete computation of all routing tables (offline mode) usually requires dozen of minutes. This is far too long to overcome link failures without interrupting running applications. The main contribution of this paper is to present a radically new approach based on a clear separation of concern for the computation of routing tables:

– ***Offline computation*** tables are computed without real-time constraint and archived for analysis and validation before being uploaded at production start-up. Several algorithms are available, both topology-agnostic and fat-tree-specific ones. Their behavior on the N-pairs communication pattern have been validated on a transaction level simulator.
– ***Online computation*** only routing tables modifications needed to bypass faults or to deal with recoveries are computed with soft real-time constraint, uploaded and archived, while still remaining (transitively) deadlock-free.

As a result, the BXI routing offline mode can compute all routing tables of a 64k nodes full fat-tree in less than 4 min on commodity hardware while the online mode can deal with at least 25 % of faults and recoveries transparently.

The BXI routing component architecture can be adapted to any other interconnect technology as it does dot depend on it. The minimal requirements are:

– Ability to compute full routing tables from a given topology (offline mode) and to upload them to all switches; this can be seen as quite basic, but performing this computation for big topologies in reasonable time frame (couples of minutes) can be challenging.
– Availability of link failure/recovery notification to a central component (`Backbone`); this notification system can be in-band as with Infiniband technology or out-of-band (using a separate management network) as with BXI technology;
– Capability of switches to receive routing tables modifications instead of full routing tables (`rtmods`); this upload can also be inband of out-of-band.

BXI provides several offline routing algorithms and two *transitively deadlock-free* [36] online routing algorithms. We assume and expect more will be proposed in the next future either to provide better overall routing quality and/or to support new topologies.

Next short term follow-up is to make several experiments with other combinations of topologies, offline and online routing algorithms along with the injection of simultaneous faults and recoveries. `Metricsd` implementation is a real challenge due to time complexity: producing metrics on an exascale topology can take a lot of time (several hours); representing these metrics is also a challenge in itself because of the amount of available data. Finally, the injection of `Metricsd` results into `Computerd` forming a control loop has not been studied yet. It is a big and interesting challenge for the next future.

Acknowledgments We are thankful to the Portals team at Sandia Nat. Lab. for their unconditional support, particularly: Ron Brightwell, Brian Barrett (now at Amazon) and Ryan Grant. We also acknowledge the passionate discussions we had with Keith Underwood from Intel during the early stages of this project. We also would like to thank our colleagues, Jean-Pierre Panziera, Ben Bratu, Anne-Marie Fourel and Pascale Bernier-Bruna for their reviews and valuable comments.

References

1. Derradji S, Palfer-Sollier T, Panziera J-P, Poudes A, Wellenreiter F (2015) The bxi interconnect architecture. In: 2015 IEEE 23th annual symposium on high-performance interconnects (HOTI)
2. Agarwal A (1991) Limits on interconnection network performance. IEEE transactions on parallel and distributed systems, vol 2, pp 398–412 (online). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.8845
3. Duato J, Yalamanchili S, Lionel N (2002) Interconnection networks: an engineering approach. Morgan Kaufmann Publishers Inc., San Francisco
4. Leiserson CE (Oct. 1985) Fat-trees: universal networks for hardware-efficient supercomputing. IEEE Trans Comput 34(10):892–901 (online). http://dl.acm.org/citation.cfm?id=4492.4495
5. Ohring S, Ibel M, Das S, Kumar M (1995) On generalized fat trees. In: Proceedings of 9th international parallel processing symposium
6. Petrini F, Vanneschi M (1997) k-ary n-trees: high performance networks for massively parallel architectures. In: Proceedings 11th international parallel processing symposium
7. Zahavi E (2010) D-Mod-K routing providing non-blocking traffic for shift permutations on real life fat trees. Technical Report CCIT Report, Tech. Rep., 2010. (online). http://webee.eedev.technion.ac.il/wp-content/uploads/2014/08/publication_574
8. Kim J, Dally WJ, Abts D (2007) Flattened butterfly: a cost-efficient topology for high-radix networks. SIGARCH Comput Archit News 35(2):126–137. doi:10.1145/1273440.1250679
9. Ahn JH, Binkert N, Davis A, McLaren M, Schreiber RS (2009) Hyperx: Topology, routing, and packaging of efficient large-scale networks. In: Proceedings of the conference on high performance computing networking, storage and analysis, ser. SC '09. ACM, New York, pp 41:1–41:11 (online). doi:10.1145/1654059.1654101
10. Kim J, Dally WJ, Scott S, Abts D (2008) Technology-driven, highly-scalable dragonfly topology. SIGARCH Comput Archit News 36(3):77–88. doi:10.1145/1394608.1382129
11. Kim J, Dally W, Scott S, Abts D (2009) Cost-efficient dragonfly topology for large-scale systems. IEEE Micro 29(1):33–40. doi:10.1109/MM.2009.5
12. Besta M, Hoefler T (2014) Slim fly: a cost effective low-diameter network topology. In: Proceedings of the international conference for high performance computing, networking, storage and analysis, ser. SC '14. IEEE Press, Piscataway, pp 348–359. (online). doi:10.1109/SC.2014.34
13. Duato J (1997) A theory of fault-tolerant routing in wormhole networks. IEEE Trans Parallel Distrib Syst 8:790–802
14. Martínez JC, Flich J, Robles A, López P, Duato J (2003) Supporting fully adaptive routing in infiniband networks. In: Proceedings of the 17th international symposium on parallel and distributed processing, ser. IPDPS '03. IEEE Computer Society, Washington, DC, p 44.1 (online). http://dl.acm.org/citation.cfm?id=838237.838493
15. Skeie T, Lysne O, Flich J, López P, Robles A, Duato J (2004) LASH-TOR: a generic transition-oriented routing algorithm. Proc Int Conf Parallel Distrib Syst ICPADS 10:595–604
16. Lysne O, Skeie T, Reinemo SA, Theiss IR (2006) Layered routing in irregular networks. IEEE Trans Parallel Distrib Syst 17:51–65
17. Flich J, Skeie T, Mejia A, Lysne O, Lopez P, Robles A, Duato J, Koibuchi M, Rokicki T, Sancho JC (2012) A survey and evaluation of topology-agnostic deterministic routing algorithms. IEEE Trans Parallel Distrib Syst 23(3):405–425
18. Cherkassky BV, Goldberg AV, Radzik T (1996) Shortest paths algorithms: theory and experimental evaluation, pp 129–174

19. Chen G, Pang M, Wang J (2007) Calculating shortest path on edge-based data structure of graph. In: Proceedings of 2nd workshop on digital media and its application in museum and heritage, DMAMH 2007, pp 416–421
20. Demetrescu C, Italiano GF (2006) Experimental analysis of dynamic all pairs shortest. ACM Trans Algorithms 2:578–601
21. Theiss Ir, Lysne O (2006) FRoots: a fault tolerant and topology-flexible routing technique. IEEE Trans Parallel Distrib Syst 17:1136–1150
22. Mejia A, Flich J, Duato J, Reinemo SA, Skeie T (2006) "Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori. In: 20th International parallel and distributed processing symposium, IPDPS 2006, vol 2006
23. Flich J, Mejia A, Lopez P, Duato J (2007) Region-based routing: An efficient routing mechanism to tackle unreliable hardware in network on chips. In: Proceedings of NOCS 2007: first international symposium on networks-on-chip, pp 183–194
24. Sem-Jacobsen FO, Lysne O (2008) Fault tolerance with shortest paths in regular and irregular networks. IPDPS Miami 2008. In: Proceedings of the 22nd IEEE international parallel and distributed processing symposium, program and CD-ROM, no. 1
25. Zahavi E, Keslassy I, Kolodny A (2014) Quasi fat trees for HPC clouds and their fault-resilient closed-form routing. In: 2014 IEEE 22nd annual symposium on high-performance interconnects (HOTI). IEEE, pp 41–48
26. Dijkstra EW (1971) A short introduction to the art of programming. Technische Hogeschool Eindhoven Eindhoven, vol 4
27. Schwiebert L, Jayasimha DN (1996) A necessary and sufficient condition for deadlock-free wormhole routing. J Parallel Distrib Comput 32:103–117
28. Schroeder MD, Birrell AD, Burrows M, Murray H, Needham RM, Rodeheffer TL, Satterthwaite EH, Thacker CP (1991) Autonet: a high-speed, self-configuring local area network using point-to-point links. IEEE J Select Areas Commun 9(8):1318–1335
29. Greenberg RI, Leiserson CE (1985) Randomized routing on fat-trees. 26th annual symposium on foundations of computer science (sfcs 1985)
30. Rodriguez G, Minkenberg C, Beivide R, Luijten RP, Labarta J, Valero M (2009) Oblivious routing schemes in extended generalized fat tree networks. In: IEEE international conference on cluster computing and workshops, 2009. CLUSTER'09. IEEE, pp 1–8
31. Kerbyson DJ, Lang M, Johnson G (October 2006) PAL Roadrunner Report 2: application specific optimization of infiniband networks. Tech Rep
32. Zahavi E (2012) Fat-tree routing and node ordering providing contention free traffic for MPI global collectives. J Parallel Distrib Comput 72(11):1423–1432. Communication Architectures for Scalable Systems (online). http://www.sciencedirect.com/science/article/pii/S0743731512000305
33. Gómez C, Gilabert F, Gómez ME, López P, Duato J (2007) Deterministic versus adaptive routing in fat-trees. In: Proceedings of workshop on communication architecture on clusters (CAC07)
34. Kim J, Dally WJ, Abts D (2006) Adaptive routing in high-radix clos network. In: Proceedings of the 2006 ACM/IEEE conference on supercomputing, ser. SC '06. ACM, New York (online). doi:10.1145/1188455.1188552
35. Underwood KD, Borch E (May 2011) A unified algorithm for both randomized deterministic and adaptive routing in torus networks. IEEE international symposium on parallel and distributed processing workshops and Phd forum, pp 723–732 (online). http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6008843
36. Jean-Noël Q, Pierre V (2013) Transitively deadlock-free routing algorithms. In: Proceedings of the 2nd IEEE international workshop on high-performance interconnection networks in the exascale and big-data era, Barcelona