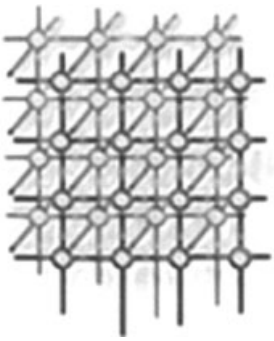
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2009)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1527



Optimized InfiniBand™ fat-tree routing for shift all-to-all communication patterns‡

Eitan Zahavi[1,*,†], Gregory Johnson[2], Darren J. Kerbyson[2] and Michael Lang[2]

[1]*Mellanox Technologies Ltd., Sha'ar Yokneam, Yokneam 20692, Israel*
[2]*Performance and Architecture Laboratory (PAL), Los Alamos National Laboratory, U.S.A.*

SUMMARY

Clustered systems have become a dominant architecture of scalable high-performance super computers. In these large-scale computers, the network performance and scalability is as critical as the compute-nodes speed. InfiniBand™ has become a commodity networking solution supporting the stringent latency, bandwidth and scalability requirements of these clusters. The network performance is also affected by its topology, packet routing and the communication patterns the distributed application exercises. Fat-trees are the topology structures used for constructing most large clusters as they are scalable, maintain cross-bisectional-bandwidth (CBB), and are practical to build using fixed-arity switches. In this paper, we propose a fat-tree routing algorithm that provides a congestion-free, all-to-all shift pattern leveraging on the InfiniBand™ static routing capability. The algorithm supports partially populated fat-trees built with switches of arbitrary number of ports and CBB ratios. To evaluate the proposed algorithm, detailed switch and host simulation models were developed and multiple fabric topologies were run. The results of these simulations as well as measurements on real clusters show an improvement in all-to-all delay by avoiding congestion on the fabric. Copyright © 2009 John Wiley & Sons, Ltd.

Received 27 March 2009; Accepted 10 August 2009

KEY WORDS: InfiniBand; fat tree; packet routing

*Correspondence to: Eitan Zahavi, Mellanox Technologies Ltd., Sha'ar Yokneam, Yokneam 20692, Israel.
†E-mail: eitan@mellanox.co.il
‡The paper was presented in the International Super Computer 2007 conference in Dresden Germany.

1. INTRODUCTION

The demand for compute cycles, for solving the ever increasing larger and more complex user problems, is met by cluster-based super computer solutions. These clusters, built for providing utility computing rather than being tailored to solve a specific problem, rely on massive parallelism for which a scalable, multipathing, high-bandwidth and low-latency network is a key. The InfiniBand™ architecture standard [1] supports all these requirements and is commonly used for building such networks. Available InfiniBand™ switches supporting up to 12 120 Gbps (or 36 40 Gbps) links and host-channel adapters (HCAs) providing 40 Gbs links are used to build fat-trees with varying ratios of cross-bisectional bandwidth (CBB). InfiniBand™ provides a deterministic static-destination routing that is computed offline and loaded into the network. Given the actual fabric topology, packet-routing tables can be computed to best fit the application and tree. Some multiple thousand compute-node installations show that application performance can be enhanced by better routing assignment and the usage of particular application communication patterns. The impact of packet routing on switching networks and its dependency on application communication patterns are well known. Multiple publications [2–4] focusing on deterministic static-destination routing for arbitrary networks provide algorithms for credit-loop-free routing, with variable degrees on resulting efficiency versus complexity. Some of them rely on the InfiniBand™ Virtual Lanes feature (sharing of physical link by several independently buffered lanes) to provide better link utilization—a hard to guarantee feature once some 'turns' are marked unusable to avoid credit loops. These generic solutions fall short of utilizing the symmetry and structure of fat-trees. Our work resolves the credit loops contingency by applying a variant of the up/down routing algorithm.

The proposed algorithm spreads the routes across a fat-tree in a way similar to that proposed by Lin *et al.* [5]. The main enhancement in our work is covering not only the well known K-ary-N-tree with arity (half the number of switch ports) limited to power of two, but extending it to the available arity of 12, CBB ratios not equal to 1, and cases where not all leaf switches are equally populated with compute nodes. Partially populated K-ary-N-trees are also supported.

Application communication pattern's effect on network congestion was extensively studied on network technologies as old as the Thinking-Machine [6] and on InfiniBand™ [7–9]. Several types of communication patterns were developed and shown to minimize network congestion. A communication pattern is defined as a sequence of communication stages. In each stage communication is between N pairs of computation nodes, where N is the number of nodes in the cluster. The communication pattern P between N pairs of nodes H_i and H_j at stage index s is noted by: $P_s = \{H_i \rightarrow H_j\}$. 'Bit flip' patterns, also known as 'Xor' patterns, are defined by $P_s = \{H_i \rightarrow H_j | j = i \oplus s\}$. It was shown that such patterns can be made congestion-free for topologies with switch arity, which is a power of two that imposes an impractical limitation for current switch devices. Another communication pattern $P_S = \{H_i \rightarrow H_j | j = i \pm s$ named 'Shift' is reported by Johnson *et al.* [10] as providing optimized application performance, and also as a super set of common 3D applications' communication patterns. Recent analysis of scientific applications [11,12] shows that the number of communication patterns are usually limited and the majority of applications exhibit regular communication patterns. Applications in this analysis included Ocean modeling (POP and HYCOM), Sea-Ice modeling (CICE), Atmospheric modeling (WRF and CAM), Shock-wave hydrodynamics

DOI: 10.1002/cpe

(SAGE and RF-CTH), and deterministic radiation transport (PARTISN, UMT2K and Sweep3D). Even when applications exhibit irregular communication patterns, they often contain regular patterns and can be mostly represented by a set of communication shifts [13]. Additionally the communicating processing-pairs, and hence the communication pattern, may be dynamic—it may change from one application iteration to the next but can persist for a reasonable amount of time before changing [11]. In this paper we describe a routing algorithm that provides a congestion-free 'shift' pattern. As shift all-to-all communication pattern is a super set of all shift patterns, it turns out that focusing on providing a congestion-free routing solution for it will improve all the above applications' performance.

The remainder of this paper is organized as follows: Section 2 introduces the variable arity fat-tree and describes an algorithm that automatically extracts its structure from an existing network. Section 3 details the proposed routing algorithm. Section 4 describes the simulation model and results for various different topologies. Section 5 provides measured application performance on 32 and 256 node clusters.

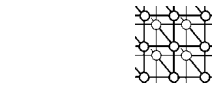
2. FAT-TREES STRUCTURE AND EXTRACTION

2.1. Introducing var-ary-*n*-trees

Fat-trees are a class of network topologies that were shown to scale their performance with the networking resources [14]. The properties and variations of fat-trees were studied in many publications [15,16]. The evolution of fat-trees started with a single common root node. A tree in which each switch is serving communication between all its branches is shown in Figure 1(a). Such trees are trivial to route as only a single minimal route exists between each pair of nodes.

However, it can be seen that traffic tends to congest when it approaches the roots of this tree as the available CBB drops by a factor of the number of children—C—on each stage. Fat-trees provide an improved topology that solves the above problem as shown in Figure 1(b). The solution is the usage of higher bandwidth links when approaching the tree roots. The fat-tree name stems from the fact that links grow fatter when approaching the root. Indeed these simple fat-trees solve the congestion at the roots, but are not practical to build as the root-to-leaf-switch total crossbar bandwidth ratio is C^N, where N is the number of levels. This creates an impractical requirement on the root switches. To overcome the impracticality of the above tree structure, a family of fat-trees named K-ary-N-trees was defined. Figure 1(c) shows a 2-ary-4-tree example. The K represents half the number of ports of each switch and N represents the number of levels of the tree. A K-ary-N-tree has N levels made out of K^{N-1} switches, each having arity of K. The K-ary-N-tree topology trades off the true non-blocking nature of its predecessors for the practical implementation using same switches for all levels of the tree. These trees are not fully non-blocking as given an oblivious packet-routing setup, it is no longer true that every permutation of source–destination node pairs can communicate without contending on the same link [9]. It is said that they are non-blocking after rearranging as it can be shown that for every permutation of source–destination node pairs, a routing solution exists such that there will be no contention. A formal definition of K-ary-N-tree was provided by

DOI: 10.1002/cpe



Petrini and Vanneschi [15]:

Definition 1. A K-ary-N-tree is composed of two types of vertices: K^N compute nodes and $N \cdot K^{N-1}$ switches. Each compute node is assigned an index $H \in \{0 \dots K^N - 1\}$. Switch nodes are assigned a tuple of the form: $S = \{r, S_{N-2} \dots S_1, S_0\}$ where $S_j \in \{0 \dots K - 1\}$ and r represents the tree level the switch is a part of. Leaf switches are at level $r = 0$. Two switches S and S' are connected if and only if $r' = r + 1$ and $S_i = S'_i \; \forall i \neq r$. The edge is labeled S_r on the switch at level $r + 1$ and S'_i on the switch at level r.

Host H_j connects to switch S if and only if $\exists p \in \{0 \dots K - 1\}$ where $j = p + \sum_{i=0}^{N-2} S_i \cdot K^i$. The connecting edge is labeled p. Given a switch tuple $S = \{r, S_{N-2} \dots S_1, S_0\}$, one can order all switches of the same level by evaluating each switch index I_S as: $I_S = \sum_{i=0}^{N-2} S_i \cdot K^i$.

Following the work done by Ohring *et al.* [16], we propose a relaxed family of trees that allows for a variable K and partial population of different levels. We enhance this work by further supporting multiple connections between switches. When multiple ports of a switch connect to the same remote switch, we say that they are a part of the same 'port group'. We name these trees var-ary-n-trees to denote that the arity might vary.

Definition 2. A var-ary-n-tree is composed of two types of vertices: compute nodes and switches. Each compute node is assigned an index $H \in \{0 \dots K^N - 1\}$. Switch nodes are assigned a tuple of the form: $S = \{r, S_{N-2} \dots S_1, S_0\}$ where $S_j \in \{0 \dots M_j - 1\}$ and r represents the tree level the switch is part of. Leaf switches are at level $r = 0$. Two switches S and S' are connected if and only

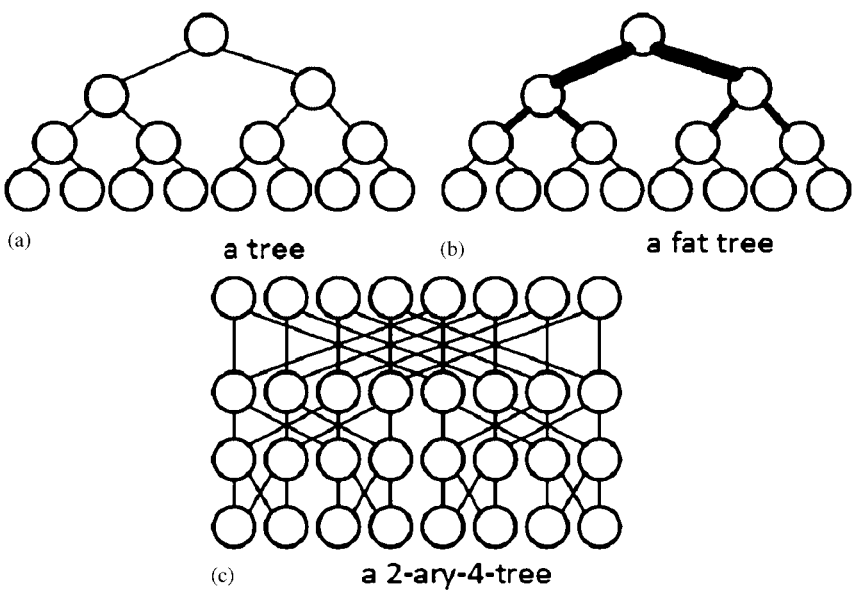


Figure 1. Three tree topologies.

DOI: 10.1002/cpe

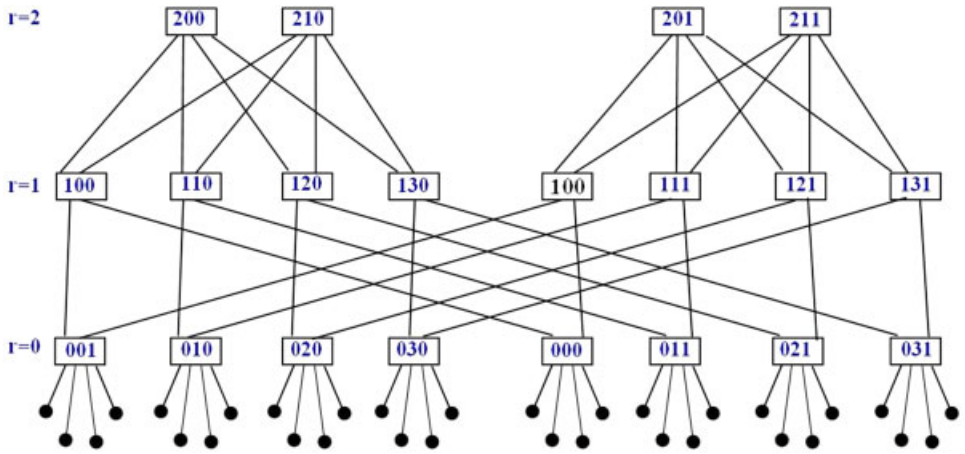


Figure 2. An example tuple assignment for var-4-2-2ary-3-tree.

if $r' = r + 1$ and $S_i = S'_i \ \forall i \neq r$. The edges in the graph represent port groups and are labeled S_r on the switch at level $r + 1$ and S'_i on the switch at level r.

We define M_j to be the number of ports groups connecting switches at level r to the lower level.

In var-ary-n-tree K may defer between the tree levels so the value each digit S_i represents is not simply K^i, instead we define P_i as:

$$P_i = \begin{cases} i = 0, & 1 \\ i > 0, & \prod_{j=0}^{i-1} M_j \end{cases}$$

and use it to define the index I_S of a switch S within its level as $I_S = \sum_{i=0}^{N-2} S_i \cdot P_i$.

Host H_j connects to switch S if and only if $\exists p \in \{0 \ldots K_0 - 1\}$ where $j = p + \sum_{i=0}^{N-2} S_i \cdot P_i$ (Figure 2).

An example for var-ary-n-tree with 3 levels with K value of 4, 2 and 2 is provided in Figure 2.

The connecting edge is labeled p. The total number of hosts is $N_H = K_0 \cdot \sum_{i=0}^{N-2} S_i \cdot P_i$.

2.2. Extracting tree structure from a given topology

Previous work [5] provided optimized routing solutions for K-ary-N-trees by specifying how to build the tree and later route it. Our work takes another approach that allows for a pre-existing topology to be routed. InfiniBand™ provides a standard management specification that enables exploring the fabric using in-band messages such that the switches, compute nodes and the links between them can be obtained. The extraction algorithm below traverses the discovered connectivity

DOI: 10.1002/cpe

graph created by performing these standard management queries. The result of the extraction algorithm is indexing assignment and edge labeling for each switch and compute node, as well as verification that the topology matches a symmetrical var-ary-n-tree that can later be routed by the algorithm proposed in Section 3.

```
start from a leaf node, push it into the BFS Queue (BFSQ)
while BFSQ is not empty
  pop a node from the BFSQ head
  foreach port of that node
    if the remote port is not a switch
        add the local port to the next available down-going port group
        continue to next port
    else
      define 'i' as the index of the tuple digit that might change between the local and
      remote nodes:
          if going up the tree from level r to r + 1 then i = r; if going down the tree from
          level r to level r − 1 then i = r − 1
      if the node on the other side does not have an assigned tuple assign it a new tuple
      such that:
          all digits with index ! = i are the same as the local node tuple; the digit 'i
            is set to the lowest number
          not causing an overlap of the tuple with other existing nodes
          AND add the node to the BFSQ
      end-if no assigned tuple on remote node
      add the local port to the port group indexed by the remote node tuple digit indexed i
      add the remote port to a port group indexed by the local node tuple digit indexed i
    end-if remote port is not a switch
  end-foreach
end-while

validate tree is var-ary-n-tree:
   foreach level require all switches have the exact same number of ports groups with exact
   same number of ports
```

Algorithm 1: Extracting var-ary-n-tree from topology.

The algorithm described in Algorithm 1: Extracting var-ary-n-tree from topology extracts the fat-tree indexing from a given topology graph. It assumes that a level is pre-assigned to each one of the fabric switches.

The complexity of the above algorithm is in the order of a single BFS through the fabric: $O(E)$ where E is the number of graph edges. This assumes an efficient unique tuple generation

DOI: 10.1002/cpe

algorithm. The current implementation uses a sequential search that is $O(N_S)$ where N_S is the number of switches.

3. ROUTING ALGORITHM

3.1. Spreading the communication

InfiniBand™ provides deterministic static-destination routing. This is accomplished by assigning each destination a local identifier (LID), and equipping each switch with a linear forwarding table (LFT). The LFT defines the out-going port through which packets are forwarded as a function of the packet destination LID. The routing algorithm task is to fill the LFT for each network switch.

The main feature of the proposed routing algorithm is spreading the routing as much as possible. For a CBB ratio of 1:1, it can be easily observed that there are enough down-going ports in each level to assign each port a single LID. Only one path going to a single LID will be flowing through each port. Shift patterns provide in-order pairing of source and destination LIDs. We have noticed that if a LID assignment for each down-going port is performed such that paths to sequential LIDs are spread across the fabric, none of the shift patterns contend. After extracting the discovered topology and allocating index tuples for each switch, the algorithm traverses the fabric in the indexing order, allocating target LIDs to each port while maintaining even port loading. CBB ratios other than 1:1 require assigning multiple LIDs per port. For the sake of equalizing the load on each port, we introduce a port 'usage' counter and always select the next less utilized port, in the indexing order. The topology traversal is performed recursively and in two stages: in stage 1, the algorithm traverses from each HCA up the tree allocating the down-going port to the LID; in stage 2, and once the down-going port is set, the algorithm assigns the LFT for the rest of the tree in the up-going direction by recursively descending down the tree.

```
foreach leaf switch (in indexing order)
    foreach compute node (in indexing order)
        obtain the LID of the compute node
        set local LFT(LID) of the port connecting to compute node
        call assign-down-going-port-by-ascending
    end-foreach
end-foreach
```

Algorithm 2: Main loop setting LFT to compute nodes.

A top-level loop running through all compute nodes and calling the LFT assignment steps is shown in Algorithm 2. The algorithm for setting the 'up' direction LFT assignment is defined in Algorithm 3. It is followed by the algorithm for setting LFT for going 'down' the tree in Algorithm 4.

DOI: 10.1002/cpe

```
Function: assign-up-going-port-by-descending
Given: a switch and an LID
foreach down-going-port-group (in indexing order)
    skip this group if the LFT(LID) port is part of this group
    skip this group if the remote node is a compute node
    find the least loaded port of the group (scan in indexing order)
    r-port is the remote port connected to it
    assign the remote switch node LFT(LID) to r-port
    increase r-port usage counter
    assign-up-going-port-by-descending r-port node
end-foreach
```

Algorithm 3: Assign up-going LFT.

To further illustrate the assign-up-going-port-by-descending algorithm, Figure 3 shows the steps of running it on switch 1.0.0 with target LID = 2. Port groups are marked by Gi. We start by examining ports P5 and P3 of G0. As port P5 is mapped as the output port for LID = 2 we ignore these ports. Port group G1 includes ports P6 and P9. We mark the number of LIDs previously routed through a port by crossing the link with one line per LID. Inspecting crosses on P1 and P2 of the switch 0.0.1, as shown on the initial stage drawing (a), we find that P2 has one less cross and thus it is used for routing packets to LID = 2 as shown on the final stage (b). The new cross is now added to switch 0.0.1 P2 to record that decision.

```
Function: assign-down-going-port-by-ascending
Given: a switch and an LID

find the least loaded port of all the groups (in indexing order)
assign the LFT(LID) of the remote switch to that port
track that port usage
assign-up-going-port-by-descending on current switch
assign-down-going-port-by-ascending on remote switch
```

Algorithm 4: Assigning down-going LFT.

The crux of the routing spreading algorithm is in the above algorithm. In Figure 4 we show how it is applied when routing to LID = 2 and the current switch is 0.0.1 that is connected to two switches 1.0.0 and 1.0.1. At the first stage, we search through the port group G0 and then G1 for the least used port. As seen on the initial stage (a), both P8 of 1.0.0 and P1 of 1.0.1 have usage count of 2. However P8 of 1.0.0 is selected as it is connected to a lower-order port group G0. The state after selection is shown in stage (b), where additional usage mark is added to the selected port and the LFT table on 1.0.0 is now set. Further steps would now invoke the algorithm for marking up-going

DOI: 10.1002/cpe

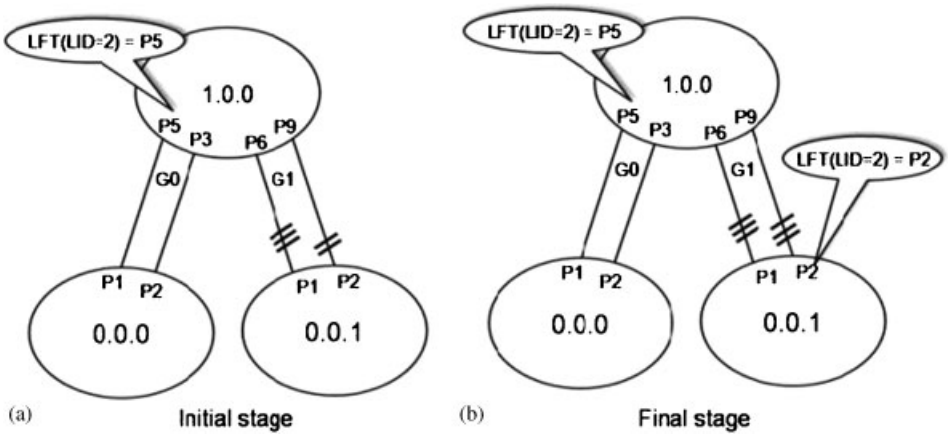


Figure 3. Demonstrating LFT assignment of up-going.

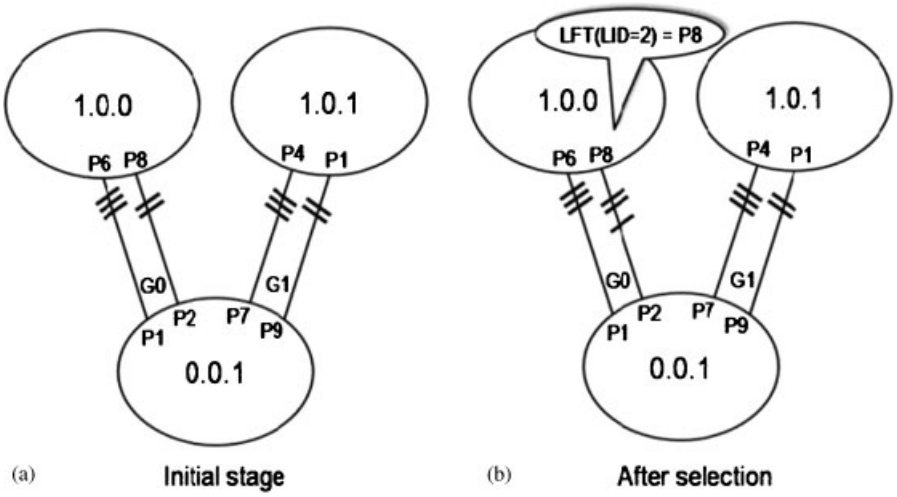


Figure 4. Illustrating down-going LFT assignment.

port LFTs for all switches connected to switch 0.0.1 and later invoking assignment of down-going ports from the selected switch 1.0.0 only.

3.2. Algorithm extension for the case of missing compute nodes

Topology-based algorithms have the weakness of being sensitive to real-life hardware failures that slightly modify the perfect symmetry of the fabric. Some of the most common deviations from vary-*n*-tree topology are missing compute nodes. To handle such cases, our algorithm uses 'place keepers' for missing nodes. During the stage of tree topology extraction, we do not require that the

DOI: 10.1002/cpe

number of compute nodes be equal between all switches. Instead, we track the maximal number of compute nodes attached to leaf switches. To maintain the tree order in case of missing compute nodes, the main loop propagates routing and sets LFT for the missing compute nodes as if they exist. Finally, when the all-to-all shift communication pattern is performed by the application, the void compute nodes are taken into account with their place in the pattern kept. For example, assume a node indexed 4 is missing from the tree and on the first communication stage of a shift pattern node-3 is supposed to talk to node-4. Instead of skipping the non-existing node-4 and having node-3 send data to node-5, node-3 does not send data to any other node. The result is an extending of the communication pattern length as if node-4 was not missing. What message length and what number of missing nodes in this approach will defeat its purpose and cost in the overall all-to-all communication time is a topic for future experimentation.

4. SIMULATION MODEL AND RESULTS

4.1. Static model

Our first tool for investigating the proposed algorithm is a static model that allows us to define arbitrary fabric topologies, assign LIDs and populate LFTs. To evaluate congestion on each stage of

Table I. Comparing static congestion obtained with fat-tree routing and OpenSM routing.

	Fat-tree		Original OpenSM	
Topology description	Worst	Average	Worst	Average
12 nodes spread on 4 IS1 connected as bipartite	1	1	2	1.4
16 nodes on 2-ary-4-tree	1	1	4	2.4
16 nodes on 4-ary-2-tree	1	1	1	1
Reordered 16 nodes on 4-ary-2-tree	1	1	1	1
16 nodes 4-ary-2-tree with merged roots	1	1	2	1.4
16 nodes on 2-ary-4-tree with merged roots	1	1	4	2.4
16 nodes on 4-ary-2-tree with merged roots	1	1	1	1
Reordered 16 nodes on 4-ary-2-tree with merged roots	1	1	1	1
32 nodes in 8 leafs—which is half 4-ary-3-tree merged roots	1	1	2	1.7
32 nodes in 8 leafs with CBB $\frac{1}{2}$ on first stage	2	1.9	4	3.2
32 nodes in 8 leafs with CBB $\frac{1}{2}$ on second stage	2	1.7	2	1.7
32 nodes in 8 leafs—which is half 4-ary-3-tree	1	1	4	2.5
48 nodes that make $\frac{3}{4}$ of 4-ary-3-tree	1	1	4	3
64 nodes 4-ary-3-tree	1	1	4	3.2
64 nodes 4-ary-3-tree with merged roots	1	1	4	3.2
64 nodes 4-ary-3-tree with merged roots	1	1	4	3.2
144 nodes 12-ary-2-tree	1	1	1	1
144 nodes 12-ary-2-tree merge roots	1	1	2	1.6
256 nodes 4-ary-4-tree	1	1	16	12
1728 nodes 12-ary-3-tree	1	1	12	11
1728 nodes 12-ary-3-tree merged roots	1	1	12	11

DOI: 10.1002/cpe

the communication pattern, the fabric is traversed from each source node to its destination according to the LFT assignments while counting the number of paths going through each link. This analysis software was built as an extension of the 'ibdm' (an InfiniBand™ data model) package that is a part of the OpenFabrics stack.

Table I provides the list of topologies that were evaluated. It compares the maximal and average contention for all stages of a shift pattern run on these topologies by the new algorithm and by the OpenFabrics Subnet Manager (OpenSM) original 'MinHop' routing algorithm.

The table shows that the proposed algorithm provides no-congestion for all cases of 1:1 CBB ratio, and constant congestion of 2 in the case of a 1:2 CBB ratio. The original OpenSM routing algorithm shows increasing congestion counts—up to 12—with the growing of network size. We model the case of a single CPU per node.

4.2. Credit propagation model

We developed a detailed simulation model for InfiniBand™ link-level flow control. The model follows Mellanox Technologie's InfiniScale™ III switch internal structure, arbitration, crossbar design and queuing mechanisms. OMNeT++ is a simulation framework used for programming the simulator. A mechanism for loading LFT tables into the simulated switches was also developed. We have simulated a 32-node topology as shown in Figure 5.

The results of simulating an all-to-all shift pattern with message size ranging from 1 to 8 K on the above network is presented in Figure 6. The simulation was run in two conditions: using 10 (single data rate—SDR) and 20 Gbps links (double data rate–DDR). Figure 6 shows up to 38% measured latency reduction for DDR and 40% for SDR. The small graph slope shows that the normalized latency slightly decreases with increase of message length. This effect can be attributed to the short congestion happening on changing of paths on message boundary. This temporary congestion is caused by the different number of hops of paths between different source destination pairs.

5. MEASUREMENT ON REAL HARDWARE

Our new fat-tree routing algorithm has been implemented and used on several production clusters. In this analysis we consider one particular system: a 256 node, 2-way dual core Opteron system interconnected using a single Voltaire ISR 9288 SDR 4x switch. The effectiveness of the fat-tree routing is compared with the original OpenSM routing algorithm for a number of cases. First, the shift pattern $P_S = \{H_i \rightarrow H_j\} | j = i \pm s$ for various shift distances is compared with the static evaluation described in Section 4. Second, the performance of two large-scale applications of interest to Los Alamos National Laboratory is evaluated.

A summary of the performance of the shift pattern for messages of size 1 MB is listed in Table II. The average time taken to perform the shift pattern across all shift distances 1–127 as well as the maximum is listed for a 256, 512 and 1024 processor execution. It can be seen that the performance of the fat-tree routing is 3× faster on average and in the worst case is over 4× faster, than when using the original OpenSM routing on a 1024 processor job. When using smaller job sizes, the performance of the original OpenSM improves slightly but also note that the performance of the

DOI: 10.1002/cpe

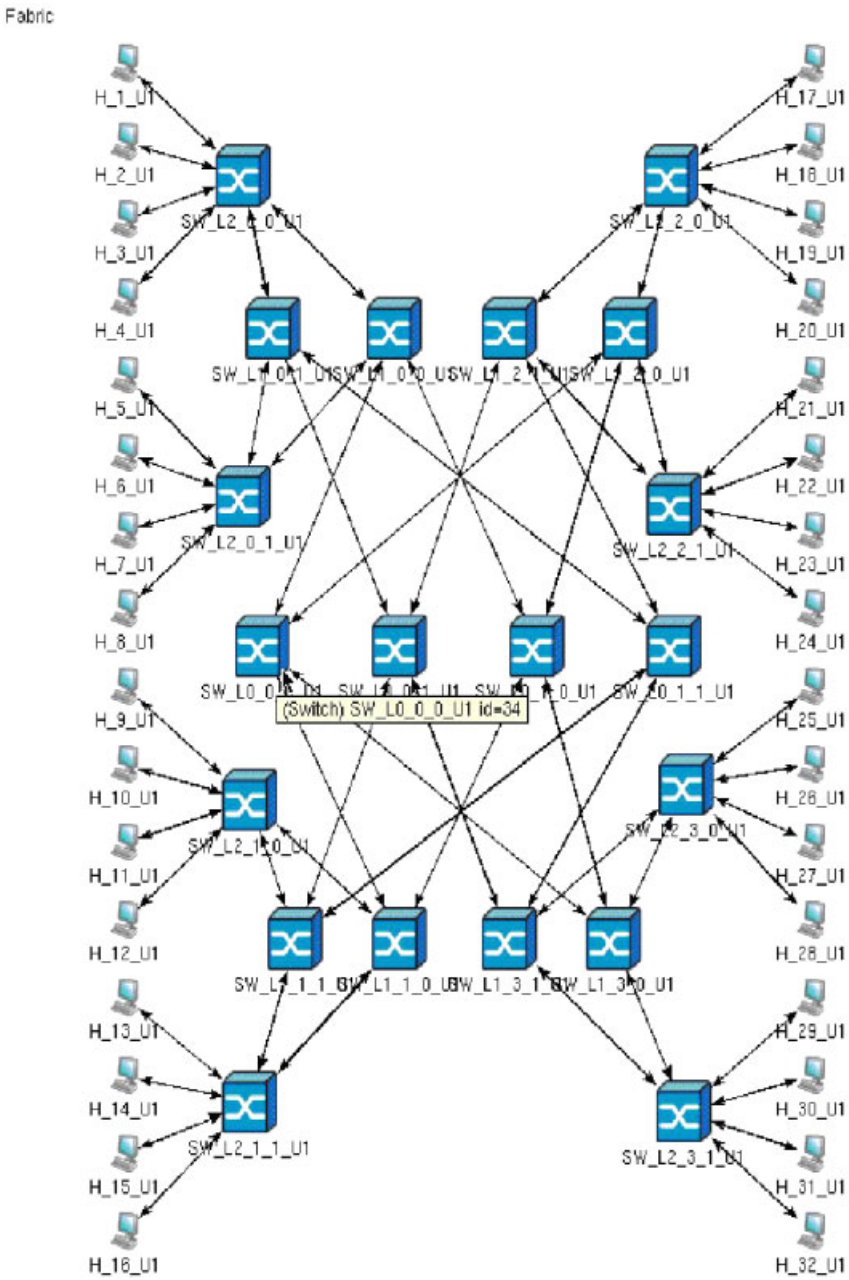


Figure 5. Simulated 32-node half 4-ary-3-tree topology (connections between switches represent two links wide port groups).

DOI: 10.1002/cpe

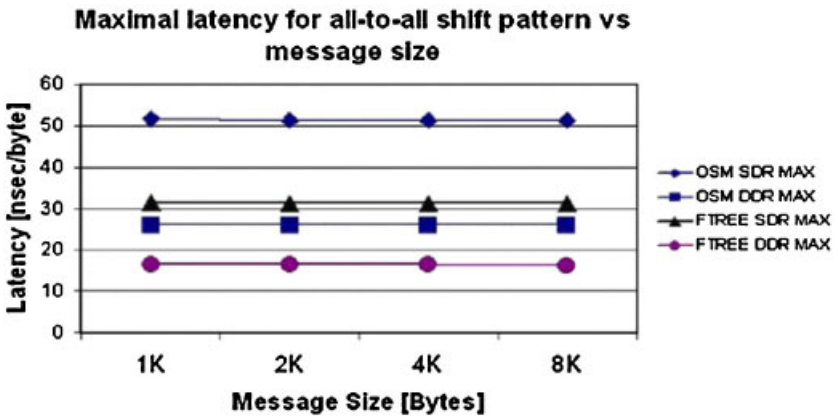


Figure 6. Normalized all-to-all shift latency.

Table II. Comparing shift pattern performance.

	Fat-tree		Original OpenSM	
Job size (# CPUs)	Worst (ms)	Average (ms)	Worst (ms)	Average (ms)
1024	6.1	5.4	22.0	16.0
512	6.1	5.4	22.0	14.7
256	6.1	5.4	22.0	13.4

fat-tree routing is constant—representing routing without contention in the network for all cases of the shift pattern.

Two large-scale applications, SAGE and Partisn, where also used to compare the performance of the fat-tree and original OpenSM routing algorithms. Unlike the case of considering a communication pattern, in which the run-time is almost all due to communication costs, the performance on a real application is part computation and part communication. Thus, the routing can only affect the communication component of the application run-time.

SAGE is a shock hydrodynamics application that typically uses 1'000s of processors at a time. It is typically run in weak-scaling mode in which the problem size per processor remains constant and the global problem size increases with processor count. It partitions the global domain in only one of its dimension that leads to an interesting scaling behavior: the boundary sizes between processors increase with scale (and hence communication volume), as well as the logical distance between communicating processors [7]. In this analysis a two-dimensional input deck that assigns 12 800 cells to each processor results in a dominant shift communication pattern with shift-distance that increases with scale (up to a distance of 24 at 1024 processors).

Partisn is an implementation of S_N transport, the solution of the Boltzmann equation using the discrete ordinates method, on structured meshes. Some details on Partisn can be found in [8]. It again is typical run in weak-scaling mode and partitions a 3-D global data domain in two of its dimensions. The input deck used in this analysis assigned a $5 \times 5 \times 400$ sub-grid of cells to

DOI: 10.1002/cpe

Table III. Performance comparison between the fat-tree and original OpenSM routing for SAGE and Partisn.

App	# CPU	Fat-tree (s)	Original OpenSM (s)	Improvement (%)
SAGE	256	0.64	0.64	0
	512	0.68	0.72	6
	1024	0.71	0.81	14
Partisn	256	106.8	108.6	2
	512	113.5	118.2	4
	1024	141.0	151.5	8

each processor. The resulting 2-D communication pattern contains two shifts, one in the logical X dimension and the other in the logical Y dimension. The change in the routing algorithm will affect on only communications in the Y dimension (a shift of $\pm S$), as the communications in the X dimension are a shift of ± 1 and are unaffected by the routing algorithm.

A summary of the performance of these two applications is shown in Table III. Here we consider the time for a single iteration of each application and show the percentage improvement on run-time when using the fat-tree algorithm. The improvement in performance is up to 14% for SAGE and 8% for Partisn. The actual improvement in just the communication components of the run-time is higher at 20 and 10% for SAGE and Partisn, respectively. We expect even higher improvements in performance on these applications at larger scale.

6. CONCLUSION AND FUTURE WORK

In this paper we introduced a new type of fat-trees: the var-ary-n-trees. We provided an algorithm to extract the tree order by traversing a discovered InfiniBand^{TM} topology. Then we described an algorithm that can be used to fill in the LFT tables of the tree switches such that a shift communication pattern is congestion free. To evaluate the algorithm, we provided static analysis of various different topologies comparing the OpenSM original routing to our proposed algorithm. The results showed no-congestion on all the trees. We further introduced a 'link-level flow control' simulator (based on OMNet++) and simulated a half 4-ary-3-tree. We measured the time it took to complete an all-to-all sequence using a shift pattern: up to 40% improvement in run-time was observed. As our work provides an algorithm for routing a single LID for each compute node, we definitely wish to extend it to the case of multiple LIDs. In the case of multiple LIDs, the fabric can be routed in multiple 'modes', and the best 'mode' may be selected either by the application or by the job-scheduler that partitions the fabric between multiple jobs. Since the first publication of this work at ISC07, the proposed algorithm was implemented in OpenFabrics OpenSM as fat-tree routing algorithm and is being used on multiple clusters.

REFERENCES

1. IBTA—InfiniBand Trade Association. IBTA Specification 1.2, vol. 1, 2004. Available at: www.infinibandta.org [Online].
2. Lopez P, Flich J, Duato J. Deadlock-Free routing in InfiniBand through destination routing. *International Conference on Parallel Processing*, *ICPP 01*, Valencia, Spain, September 2001; 427–434.

DOI: 10.1002/cpe

3. Skeie T *et al.* LASH-Tor: A generic transition oriented routing algorithm. *Parallel and Distributed Systems*. IEEE: New York, July 2004.
4. Sancho JC, Robles A, Flich J, Lopez P, Duato J. Effective methodology for deadlock-free minimal routing in InfiniBand networks. *International Conference on Parallel and Distributed Processing ICPP*. IEEE: New York, August 2002; 48–57.
5. Lin X-Y, Chung Y-C, Huang T-Y. A multiple LID routing scheme for fat-tree-based InfiniBand networks. *International Parallel and Distributed Processing Symposium* (*IPDPS'04*). IEEE: New York, 2004; 1–13.
6. Bolding K, Synder L. Congestion-free routing on the CM-5 data router. *First International Workshop PCRCW* (*Lecture Notes in Computer Science*, vol. 853), Heller S (ed.), Seattle, Washington, May 1994. Springer: Berlin, 1994; 176–184.
7. Kerbyson DJ, Alme HJ, Hoisie A, Petrini F, Wasserman HJ, Gittings ML. Predictive performance and scalability modeling of a large-scale application. *SC'01*, Denver, CO, 2001.
8. Baker RS. A block adaptive mesh refinement algorithm for the neutral particle transport equation. *Nuclear Science and Engineering* 2002; **141**(1):1–12.
9. Hoefler T, Schneider T, Lumsdaine A. Multistage switches are not crossbars: Effects of static routing in high-performance networks. *IEEE International Conference on Cluster Computing*, Tsukuba, Japan, 2008; 116–125.
10. Johnson G, Kerbyson DJ, Lang M. Application specific optimization of infiniband networks. *PAL Roadrunner*, *LAUR 06-7234*, LANL, October 2006.
11. Barker KJ, Benner A, Hoare R, Hoisie A, Jones AK, Kerbyson DJ, Li D, Melhem R, Rajamony R, Schenfeld E, Shao S, Stunkel C, Walker P. On the feasibility of optical circuit switching for high performance computing systems. *SC05*, Seattle, 2005.
12. Kamil S, Pinar A, Gunter D, Lijewski M, Oliker L, Shalf J, Skinner D. Reconfigurable hybrid interconnection for static and dynamic scientific applications. *Tech Report 60060*, Lawrence Berkeley National Laboratory, 2006.
13. Kerbyson DJ, Barker KJ. Automatic identification of application communication patterns via templates. *International Conference on Parallel and Distributed Computing Systems* (*PDCS*). Las Vegas, NV, U.S.A., 2005.
14. Leiserson CE. Fat-trees: Universal networks for hardware-efficient supercomputing. *Transactions on Computers* 1985; **34**(10):892–901.
15. Petrini F, Vanneschi M. k-ary n-trees: High performance networks for massively parallel architectures. *Parallel Processing Symposium IPPS97*, Geneva, Switzerland, 1997; 87–93.
16. Ohring SR, Ibel M, Das SK, Kumar MJ. On generalized fat tree. *Proceedings of the Ninth International Parallel Processing Symposium*, Santa Barbara, CA, 1995; 37–44.

DOI: 10.1002/cpe