Deadlock-free Routing Based on Ordered Links

Dah Ming Chiu, Miriam Kadansky, Radia Perlman, John Reynders†,*
Guy Steele, Murat Yuksel‡†
Sun Microsystems Laboratories, Burlington MA
dahming.chiu@sun.com, miriam.kadansky@sun.com, radia.perlman@sun.com, guy.steele@sun.com

Abstract

This paper describes a new class of deadlock-free routing algorithms for irregular networks based on ordered links. In this case, the links are ordered by partitioning them into a set of ***layers****, each layer containing a spanning tree (when possible). Deadlock free routes can then be derived by using links with non-decreasing order. The deadlock-freedom property is proved. Two different implementations of the routing algorithm are studied. The resultant performance of these algorithms is then compared to other known algorithms: the shortest-path algorithm (which may result in deadlocks) and the up*/down* algorithm. Various performance metrics are considered, including path-length, network capacity, fault tolerance and time of computation. We argue that network capacity is the most important metric to optimize. It is shown that the proposed algorithms are promising since they usually achieve higher network capacity than up*/down*, while they perform only slightly worse than up*/down* in other metrics.*

1. Introduction

Recently interest has increased in building large-scale data center computer systems interconnected by a switched fabric (network). Infiniband(tm)[2] is one example of an industrial standard of high-speed interconnect for this purpose.

It is common in such networks to use hop-by-hop flow control rather than dropping packets. Therefore it is necessary for routing to be *deadlock-free*. In traditional, e.g., IP-based networks, routing is just required to be loop-free, which means that any individual route is loop-free. Deadlock-freedom is an additional requirement whereby routes which are individually loop-free do not interact in a way that can cause deadlocks without dropping packets.

Since the routing table is computed centrally, there is the opportunity to make use of multiple (equal-cost) paths for load-balancing, maximizing overall system throughput. This paper describes a new routing algorithm that addresses these requirements.

The organization of the paper is as follows. In section 2, we first discuss relevant past work on this topic. In section 3, we formally define the deadlock condition. In section 4, we describe our algorithm and various methods of computation. Section 5 contains a performance evaluation of our algorithm, and comparison with other algorithms in terms of average path length and computation time, for a set of test topologies. In section 6, we discuss how to make use of alternative (equal-cost) paths for network optimization. This step can be applied to various routing algorithms if they are computed centrally. Finally, we conclude and discuss future work.

2. Past Work on Deadlock-free Routing

Earlier approaches to avoid deadlocks rely on using regular topologies (such as hypercube or torus). Simple and fixed routing rules are known to be deadlock-free for such topologies and can be readily applied. For a (random) irregular topology, however, a deadlock-free routing function must be computed by a routing algorithm, and there are very few such algorithms known.

One important deadlock-free routing algorithm is the up/down algorithm [6]. The up/down algorithm first establishes a *direction*, either *up* or *down* for each unidirectional link[1] in the network. Then the deadlock-free routing is computed based on the constraint that each route must not contain a *down* link before any *up* link. An important property of the up/down algorithm is that it can be computed in a distributed fashion, and it is very simple.

*†Reynders contributed to this work while he was at Sun Labs. He is now at Celera Genomics.

†‡Yuksel contributed to this work while he was an intern at Sun Labs during the summer of 2001. He is now a graduate student at Rensselaer Polytechnic Institute.

[1] In most networks, links can be used for bidirectional communication. Each bidirectional link should be considered as two unidirectional links for our discussions.

0742-1303/02 $17.00 © 2002 IEEE

Subsequently, [10] describes a technique that uses alternative routes and relies on one subset of the alternatives (for each source and destination) to be deadlock-free to guarantee the routing function to be deadlock-free. This technique can be used to optimize the performance of known deadlock-free routes. This technique, however, requires the switches to support multi-path forwarding, which is not universally true. For example, Infiniband does not support multi-path forwarding by the switches. In this paper, we assume switches are *dumb* and only hold one next-hop for each destination.

Dally and Seitz in their classic paper [4] established a necessary and sufficient condition for deadlock-free routing - there must be no cycles in the channel dependency graph (to be defined in 3.2 below). In the proof, they deduced that an acyclic channel dependency graph implies a total ordering of channels (links), and deadlock cannot result when each route uses channels in ascending order.

Our routing algorithm applies and extends the Dally-Seitz result to avoid deadlocks, and in addition searches for routes that optimize other metrics, such as shortest path length and maximum capacity.

3. Basic Concepts

3.1. Network and Routing Table

The network is a directed graph denoted by

$$G_{network} = (N, C)$$

where N is a set of nodes and C is a set of (unidirectional) links [2] (called channels in [5]). A routing algorithm computes an $N \times N$ *routing table*, R, where R_{ij} stores the link used by node i as the first hop in its route to node j. When alternative routes are considered, each R_{ij} stores the list of alternative first hops on equal-distance routes to node j. The routing table is equivalent to the routing function in [5]. A companion matrix to R is the (shortest) distance matrix S, where each entry S_{ij} is the distance between node i and j by using the path according to the routing table.

A basic requirement is that R must be loop-free. Intuitively, a sufficient condition to ensure loop-freedom is to only allow shortest path routes in R. There exist many algorithms to compute shortest paths, for example, the Bellman-Ford algorithm, Dijkstra's algorithm and the Floyd-Warshall algorithm[3].

Deadlock-freedom is a new requirement on R. This is developed in the next section.

[2] Throughout this paper, we assume each link has a unit cost. It is not hard to extend most of our results to the case where links have different positive costs since they are equivalent to multiple unit cost links.

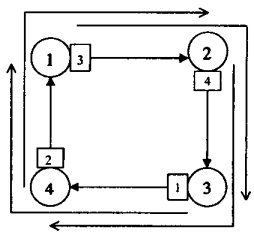

Figure 1. Example topology and routes to illustrate a deadlock condition

3.2. The Deadlock Condition

Each node has a finite number of buffers for each outgoing link. These buffers make each link two unidirectional links (in opposite directions). A deadlock can occur when a number of routes, each traversing more than one link, have paths overlapping to form a cycle. Consider Figure 1 for example. There are four nodes, 1, 2, 3 and 4, connected by eight links (1,2), (2,1), (2,3), (3,2), (3,4), (4,3), (4,1) and (1,4). The four routes 1→2→3, 2→3→4, 3→4→1, 4→1→2 are all minimum in path length, but form a cycle and hence can produce a deadlock. In Figure 1, only the links utilized by the routes are shown, together with the buffers associated with the links. The number in each buffer indicates the destination node for all the packets in the buffer during the deadlock condition. Since all the buffers are filled and none of the links can make any progress in forwarding packets, we have a deadlock. This is a standard example. For more discussions see pages 73-74 of [5].

More formally, the deadlock condition can be defined in terms of a cycle in the link dependency graph (in [5], this is called the *channel* dependency graph):

$$G_{dependency} = (C, E)$$

Note, the vertices of the $G_{dependency}$, C, corresponds to the set of (unidirectional) links in $G_{network}$, and E is the set of link dependencies defined by the routes in R as follows. When i and j are two consecutive links in a route in R, then an edge (i, j) is an element of E. Clearly, $G_{dependency}$ is also a directed graph.

There are a number of algorithms to check for cycles in a directed graph. One way is by using the adjacency matrix D of $G_{dependency}$. The Floyd-Warshall algorithm[3] can be used to compute the transitive closure of D, denoted D^*. There is a cycle in the link-dependency graph if and only if some of the diagonal elements of D^* are non-zero.

Even with this precisely defined condition, it is not practical to keep checking it while constructing shortest path routes, as the computation cost is quite high. The approach taken below, is therefore to find some constraint that prevents cycles from forming during the construction of shortest path routes. Such an approach may not derive the optimal routing table, but is much faster.

4. Deadlock-free Routing Based on Ordered Links

4.1. Deadlock-freedom and Ordered Links

We now establish another equivalent condition for deadlock-free routes which can be more readily used in a routing algorithm.

Let us assign a unique *rank* to each link in the network, so that the links are ordered. If the successive links used by each route have increasing ranks, then these routes cannot cause a cycle in the link dependency graph. Let us assume there is a cycle formed by these routes; and let us start with one of these routes in the cycle and follow the links; and then follow the links of the route that overlap with the current one, and so on. The ranks of these links must continue to increase; but when we get back to a previously used link (since there is a cycle), it implies that the rank of the very first link is higher than the other links traversed so far - a clear contradiction.

Further, we note that the inverse must also be true, namely, if a set of routes does not form a cycle in the link dependency graph, then there must exist an ordering of links such that the links used by each route follow an increasing sequence. This can be proved constructively (start with a set of routes and assign order to links). Since there are no cycles, there must exist at least one route whose first link does not overlap with other routes. Start with this link and do a *topological sort* by following the subsequent routes. It should be intuitive that the lack of cycles in the dependency graph means the procedure will terminate with an ordering of all links used in the routes.

These statements essentially prove the following result originally established by Dally and Seitz.

Theorem 1 *A set of routes is deadlock-free if and only if there exists an ordering of the links such that all the routes use links in increasing order.*

Although this theorem does not by itself construct deadlock-free routes, we show how it can be used to develop algorithms that do.

4.2. The Layered Algorithms

The links of $G_{network}$, C, can be partitioned to form a set of subnetworks

$$G_i = (N_i, C_i) \qquad i = 1, ..., m$$

such that

$$C_i \bigcap C_j = \emptyset \qquad \forall i, j \quad i \neq j$$

$$\bigcup_{i=1}^{m} C_i = C$$

Further, we can consider these subnetworks as ordered *layers* of the original network, where for example G_1 is the lowest layer, G_2 is the next layer up and so forth.

The basic idea is decomposition. We decompose the network into small enough subnetworks such that we know how to compute deadlock-free routes for the subnetworks.

Consider the following algorithm, to be referred to as a *layered* algorithm:

1. Partition the links of $G_{network}$ into a set of ordered layers.
2. Compute a deadlock-free route for each pair of nodes (if connected) in each layer.
3. Find better routes by possibly concatenating routes from different layers, subject to the constraint that the links in each route must belong to layers in a non-descending order.

Before we explain how each step is done, let us first establish that this algorithm will produce deadlock-free routes.

Following theorem 1, we can assign a rank to each link in a layer after step 2 such that each route computed in step 2 will consist only of links in non-descending order. If we obey the constraint in step 3, the optimized routes will also consist of links in non-descending order. Again due to theorem 1, we know the optimized routes (after step 3) are deadlock-free.

Corollary 2 *The routes produced by the layered algorithm are deadlock-free.*

It is important to note that the layered algorithm no longer guarantees inclusion of all possible deadlock-free routes, hence it may not be optimal. We will have much more to say about this later when we evaluate the algorithms.

Secondly, depending on the partitioning of $G_{network}$ (step 1) used, the layered algorithm cannot guarantee it will compute a deadlock-free route for every pair of nodes in

step 2. In other words, it does not guarantee the routing table (routing function) is *connected*[5]. This problem, however, can be easily fixed.

A sufficient condition to ensure the routing table is connected is to require that at least one of the layers contains a route for each pair of nodes. We call such a layer a *spanning* layer.

4.3. Partitioning into Ordered Trees

We now consider a specific way of doing step 1 of the layered algorithm, called ordered trees partitioning. Let us partition $G_{network}$ by successively removing links from it to form a minimum spanning tree. There are a couple of points worth clarifying. First, the notion of a spanning tree is based on bidirectional links spanning the set of nodes in a graph. So, if there is a bidirectional link included in a spanning tree, we are including both unidirectional links between the same pair of nodes. Secondly, it is clear that sooner or later we will not have enough links in the remaining graph to form a partition that is a spanning tree. In that case, what is extracted from the remaining graph is a maximally spanning forest; which means we extract as many links as possible without forming any loops.

A well-known algorithm - Kruskal's minimum spanning tree algorithm[3] - can be applied to partition the graph $G_{network}$ into spanning trees. Kruskal's algorithm extracts a minimum spanning tree from a graph. The ordered trees partitioning is done by applying Kruskal's algorithm repeatedly until all the links in the original graph are exhausted. Kruskal's algorithm is very efficient ($O(e)$, where e is the number of links).

The basic motivation for organizing the layers as spanning trees (and forests) is so that the routing table is connected.

Kruskal's algorithm extracts the minimum spanning tree based on the order of the links given to the algorithm. An interesting question is whether given a topology it is advantageous to try to put certain links in as low a layer as possible. In particular, certain links seem to be connecting nodes that are a *hub* for the rest of the nodes. We experimented with a scheme that ranked links according to this weight before giving them to Kruskal's algorithm. The results will be discussed later.

Since we are going to consider only ordered tree partitions, we call the layered algorithm the *ordered-tree* algorithm for deadlock-free routing. We also call the corresponding deadlock-free condition the *ordered-tree condition.*

4.4. Routing Table computation

The next task is to compute the routing table, satisfying the constraint that each route uses links only from layers of non-descending order (i.e. ordered-tree condition). This can be done by adapting various *shortest paths* algorithms to satisfy our ordered-tree condition.

We describe two such algorithms that we studied. One is to adapt the dynamic programming algorithm for all-pairs shortest paths via matrix multiplication[3]. We refer to the resultant algorithm as the All Pairs Ordered Tree (APOT) algorithm. The other is an adaptation of the Bellman-Ford algorithm[4]. This is referred to as the Bellman-Ford Ordered Tree (BFOT) algorithm. The latter algorithm can be adapted for distributed computation as well.

Note, in most scenarios where deadlock-free routing is required, the routing table computation is carried out centrally by a network manager. The resultant routing information is then downloaded to the switches. This is why we are mainly considering centralized algorithms.

4.4.1 APOT. In the regular all-pairs matrix multiplication, in each iteration a new $n \times n$ distance matrix is computed, each entry of which represents a path with at most t hops (where t is the iteration number and n is the number of nodes).

In APOT, each intermediate result is an $n \times n \times m$ matrix, Q, where q_{ijk} represents the cost of a path from i to j using links up to layer k, with at most t hops (where t is the iteration number, and m is the number of layers).

The pseudo code of APOT is shown in Figure 2.

Note, P is a working matrix used in each iteration to hold the result for the t^{th} iteration while Q is holding the result from the $(t-1)^{th}$ iteration. The variables i, j, and k have their equivalents in the regular all-pairs matrix multiplication algorithm. The variable l references the l^{th} layer entry of the intermediate result; and the variable h iterates through all layers higher than l.

The pseudo code to compute the shortest routes and alternative routes follows the same logic. It is not included as it does not shed more insight on the complexity.

In [3], it is shown that the complexity of all-pairs matrix multiplication is $O(n^4)$. For the APOT algorithm, the intermediate result has $n \times n \times m$ elements and each element takes $n \times m$ steps to compute. So each iteration takes $O(n^4 m^2)$ steps. Roughly, m is e/n (where e is the number of links). Therefore, APOT's complexity is $O(n^2 e^2)$. It is very compute intensive.

4.4.1 BFOT. Here, we start with describing the distributed version of the Bellman-Ford Ordered Tree algorithm, which

[3] a clear description can be found in Chapter 26.1 of [3].

[4] a clear description can be found in Chapter 25.3 of [3].

```
for (t=0; t<n; t++) {
  init(P);
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      for (l=0; l<m; l++) {
        for (k=0; k<n; k++) {
          for (h=1; h<m; h++) {
            distance =
              Q(i,k,l) + S(k,j,h);
            if (distance < P(i,j,h))
              P(i,j,h) = distance;
          }
        }
      }
    }
  }
  Q = P;
}
```

Figure 2. Pseudo code for APOT

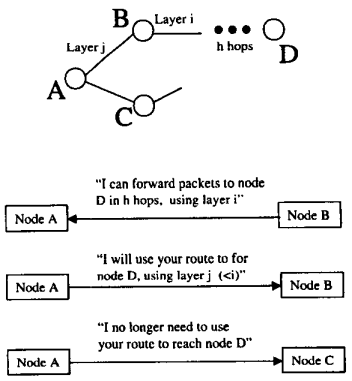

Figure 3. Example protocol exchange between node A and its neighbors B and C, in which node A originally routes to node D via C, but discovers a new route to node D via B

is more general. The algorithm is best visualized from the computation of best route and distance from a single node's point of view.

Initially, a node A knows the shortest path to reach all its neighbors (which by definition is one hop). The neighbors all tell node A their shortest distance to reach the rest of the nodes, using the following message:

route-info(destination, distance, layer)

The **route-info** message says there is a path from the neighbor (that sends this message) to *destination* using links in the given *layer* or higher, with the given *distance*.

When multiple legal routes to reach a destination are found, each node keeps all *equi-distance* shortest-distance routes as alternatives. When informing its neighbors, however, only the alternative that is the least constraining (i.e. the layer of the first-hop link is highest in comparison to the other alternatives) is sent in the **route-info** message[5].

When node A receives a **route-info** message from its neighbor, it first determines if it helps to produce a shorter route (satisfying the ordered-tree condition) than the current set of alternative routes. If not, the **route-info** message is ignored. If the information given in the **route-info** message produces a new (alternative) route to the *destination* (satisfying the ordered-tree condition), then node A must notify the neighbor that sent the **route-info** message using the following message:

route-constrain(destination, layer)

Node A sends the **route-constrain** message to the neighbor, say B, that has been chosen to forward node A's packets for the given *destination*. Node B will be called the *forwarding neighbor* for the given destination.

Node B, upon receiving the **route-constrain** message from node A, must check to see if any of the alternative routes to reach the *destination* uses a link in a *layer* lower than that given in the **route-constrain** message. If so, those alternatives must be marked as *unusable* together with a list of neighbors that rendered the route unusable. This list is called a *constraint list*. This ensures the ordered-tree condition is not violated.

Upon adopting a new route, node A (in addition to sending the **route-constrain** message) must itself send (additional) **route-info** messages to other neighbors about its newly discovered route for *destination*. Further, if a forwarding neighbor for the given destination will no longer be used, the following message is sent:

route-constrain(destination, 0)

This is used to tell the forwarding neighbor that the constraint previously imposed on it is no longer necessary. This is why those alternative routes that violate the constraint are only marked as unusable rather than discarded. Upon the receipt of the above unconstraining message, the corresponding routes previously marked unusable will have their corresponding constraining neighbor removed from the constraint list. When the constraint list is empty, an unusable route becomes usable again.

All other nodes in the network follow the same procedure as node A. Figure 3 shows an example protocol exchange between node A and its neighbors. In this case, node A originally uses node C to forward packets to node D. Upon receiving a **route-info** message from node B, it switches to

[5] This is an optimization that is carried out when one or more constraining alternative route(s) are subsequently discovered

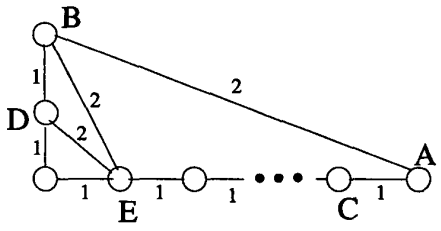

Figure 4. Example topology when BFOT does not compute optimal routes

use node B to forward packets to node D. Note, this is possible because the link from A to B is in layer j, which is lower than i, the layer B starts with when forwarding packets to D.

Other than the **route-constrain** messages, this is basically the Bellman Ford algorithm which is known to converge and terminate, assuming the protocol exchanges are reliable. In a distributed implementation of BFOT, it will take additional engineering to ensure the robustness and convergence of the protocol.

In our case, we had a centralized implementation of BFOT, so there are no communication reliability issues. All the messages, together with the identities of sender and receiver, are put on a centralized queue. The algorithm terminates when the centralized queue empties.

The computational complexity of Bellman-Ford is known to be $O(ne)$ for a single source, and therefore $O(n^2e)$ for all source and destinations. BFOT introduces the additional processing due to the **route-constrain** messages, which are used

1. in response to a **route-info** message
2. in correcting an earlier **route-constrain** message

Neither of these uses exceed the number of **route-info** messages. Therefore the computational complexity of BFOT remains $O(n^2e)$. It is a factor of e better than APOT.

The downside, however, is that BFOT does not necessarily compute the optimal routes. This can be illustrated by the example in Figure 4.

The number next to each link is the layer the link (both unidirectional links) belongs to. There are nine links between node C and E, all in layer 1. Let us focus on what happens at node A, in searching for a route to node D. Node A receives **route-info** messages from its neighbors B and C:

route-info(B, D, 1, 1)
route-info(C, D, 10, 1)

Although the route through B to D is shorter, it violates the layered rule. So node A has to reach D via C, a route that takes 11 hops.

With a more global view, one can see that node B has another route to D which uses layer 2 links and is 2 hops long. Although this is not the shortest route from node B to node D, it can dramatically cut down the cost for node A to reach node D. If node B offered this to A, however, node B will get a **route-constrain** from node A and node B's route to node D will also become 2 hops. This is a rather small sacrifice for node B and will produce lower average path lengths. It is this kind of trade-off, which requires more global considerations, that is missed by the BFOT algorithm.

Nonetheless, as will be shown in the next section, for many topologies we considered, the BFOT's average path length is almost as good as the optimum computed by APOT.

5. Comparisons of Path Length and Computation Time

In this section, we compare the two implementations of the ordered tree algorithms, APOT and BFOT, to the unconstrained shortest path, OPT (that may have deadlocks) and the up*/down* algorithm, UD (which is deadlock-free).

We used IRFlexSim[1], a public domain analysis and simulation tool for deadlock-free routing algorithms produced by the SMART group of USC, for much of our evaluation. We added our implementations of BFOT and APOT and various other algorithms for utilizing alternative paths. We relied on IRFlexSim for topology generation, computation of the unconstrained optimal routing function, and up*/down* routing function.

We noticed that the equal-cost alternative routes in the up*/down* routing table may contain routes that produce deadlocks. One way to avoid deadlocks is to use switches that can enforce the up*/down* rule in real time. Alternatively, we added a function to remove those alternative routes that violate the up*/down* rule from the routing table. Later, we found out that this problem had also been discovered by Sancho, Robles and Duato [7]. They did a thorough analysis and concluded that removing the deadlock-causing alternative routes only slightly affects the average path length of the up*/down* algorithm.

We compared algorithms, denoted OPT (optimal), UD (up*/down*), APOT and BFOT. The extra column BFOT+ is the same as BFOT except we sorted the links[6] before running them through the Kruskal's algorithm repeatedly to

[6] In this particular case, we sorted the links as follows. First rank all the nodes based on the number of links connecting the node; then rank the links by adding the rank of the two nodes at the end of each link. This metric tries to put the more highly connected links in the lowest partitions.

partition the network. BFOT+ was included to see the effect of different partitionings.

To compare the different algorithms, we used the topologies listed in Table 1. The names of the topologies roughly follow the convention:

type-nodes-links-otherparam...

Most topologies have the number of nodes and links encoded in the topology name. For type=random, the other parameters are: minimum and maximum degree, and a seed for the random number generator. Only those topologies with a *dup* in it have duplicate links (between a pair of nodes).

The topology *rndm-nice* has 16 nodes and 29 links without duplicate links; we gave it that name because it was the first random topology we used for which BFOT+ performed particularly well.

Table 2 summarizes the comparison, in terms of the total time to compute for all topologies, and average path length (normalized by the optimal average path length).

Let d_{OPT}, d_{UD}, d_{APOT} and d_{BFOT} denote the average path lengths of the four algorithms, and t_{OPT}, t_{UD}, t_{APOT} and t_{BFOT} denote the time of computation for these algorithms respectively. As expected,

$$d_{OPT} \leq d_{APOT} \leq d_{BFOT}$$

$$d_{OPT} \leq d_{UD}$$

In addition, we note the following:

1. d_{UD} is usually somewhat better than d_{APOT} and d_{BFOT}, but it is not always true. The differences between all these values are rather small (within 10% or so).

2. The difference between d_{APOT} and d_{BFOT} is really small. This confirms our belief that the suboptimality of d_{BFOT} is not a problem in practice.

3. The effect of sorting the links ($d_{BFOT+} - d_{BFOT}$) seems to be more significant than the difference between BFOT and APOT. Similar effects are observed when picking a different node as the root in the up*/down* algorithm, as noted in [9].

Based on the path length metric, there is no strong reason for using any one particular algorithm.

In terms of computation time, we observe

$$t_{OPT} < t_{UD} < t_{BFOT} < t_{APOT}$$

The algorithms for computing OPT, UD and BFOT (and BFOT+) are all derived from the Bellman-Ford algorithm. This explains why their computation times are quite similar. But t_{APOT} is significantly higher than the other three. This is also expected, as we showed the computational complexity of APOT is worse than BFOT by a factor of e.

Table 1. Network topologies

Random topologies	regular topologies
rndm-16-24-1-8-4005	hypercube-32-80-5
rndm-16-25-1-8-4004	hypercube-64-102-6
rndm-16-26-1-8-4003	ring-32-32
rndm-16-27-1-8-4002	ring-64-64
rndm-16-28-1-8-4001	torus2-16-32-4-4
rndm-16-32-1-8-3001	torus2-64-128-8-8
rndm-16-32-1-8-3002	torus3-16-32-2-2-4
rndm-16-32-1-8-3003	torus3-32-72-2-4-4
rndm-16-32-1-8-3004	torus3-48-116-3-4-4
rndm-16-32-2-6-3009	torus3-64-160-4-4-4
rndm-16-32-2-6-3010	
rndm-16-32-2-6-3011	
rndm-16-32-2-6-3012	
rndm-16-48-1-8-3013	
rndm-16-48-1-8-3014	
rndm-16-48-1-8-3015	
rndm-16-48-1-8-3016	
rndm-16-48-d-5-5003	
rndm-16-48-d-9-5001	
rndm-16-48-d-9-5002	
rndm-16-48-d-9-5004	
rndm-32-96-1-8-1005	
rndm-32-96-1-8-1006	
rndm-32-96-2-8-1007	
rndm-32-96-2-8-1008	
rndm-64-128-1-8-2001	
rndm-64-128-1-8-2002	
rndm-64-128-1-8-2003	
rndm-64-128-1-8-2004	
rndm-64-160-1-8-2009	
rndm-64-160-1-8-2010	
rndm-64-192-1-8-2013	
rndm-64-192-1-8-2014	
rndm-64-192-1-8-2015	
rndm-64-192-1-8-2016	
rndm-nice	

Table 2. Comparison of optimality and compute times of various algorithms

	Total Time	Path length (normalized)
Optimal	0.42	1.0
Up*/Down*	2.83	1.057
Layered (MM)	2891.86	1.096
Layered (BF)	6.49	1.102

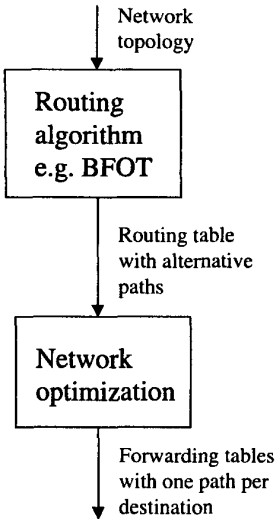

Figure 5. The network optimization step

Based on these results, we chose to use BFOT instead of APOT. Solely based on average path length and time of computation, however, the ordered tree algorithm does not show any advantage over UD.

6. Alternative Paths & Network Optimization

In high-speed networks where deadlock-free routing is required, the design center is usually aimed at using dense interconnections to increase total throughput and fault-tolerance. This is the case for Infiniband(tm)[2], for example. These are the metrics when evaluating the performance of Autonet[8], and other high-speed networks[5] as well.

In this section, we consider how these metrics can be optimized. This is done as a separate step, after the routing algorithm has computed the shortest paths as shown in Figure 5.

Given any source and destination, there are typically several alternative (equal-distance) shortest paths[7]. The ability to optimize for throughput and fault-tolerance lies in the way these alternative paths are used.

6.1. How Alternative Paths Are Used

In one paradigm, information about alternative paths is contained in the switch's forwarding table. In this case, the actual forwarding decision can be made at the time of forwarding, based on locally observed, but dynamic network conditions. The drawback of this approach is that the dynamic information is local, and it is not clear decentralized decision-making necessarily converges. Due to such concerns, randomized forwarding is often adopted.

In the other paradigm, each switch's forwarding table contains only one pre-selected path for each destination. In this case, the selection of which alternative path to include in the forwarding table is pre-computed as part of the routing algorithm. The drawback here is that there is no dynamic input. The computation has to assume certain traffic patterns exist. The advantage is that the centralized computation can take into consideration various metrics for network optimization.

For Infiniband, the switches store only one path for each destination (the latter case). This is done presumably to simplify the switch. Adaptive use of multi-paths based on dynamic information can still be tried at the source. This requires multiple addresses be given to each host, one corresponding to each different path. How to take advantage of this approach will be future work.

In this paper, we focus on the problem of how to preselect a path for the forwarding table that optimizes the following network metrics:

- network capacity
- fault tolerance

6.2. Maximum Flow and Capacity

When the network workload is given in terms of a set of flows, one may formulate a network problem to maximize the throughput of all the flows. In the abstract form[8], the problem is known as the *maximum flow* problem. This problem is often considered for road traffic routing, for example.

We focus on the case when the flows are not known. The first question is how to generate a random set of flows for a given (random) topology.

We consider the given network topology the *switch* topology, and generate a given number of *endnodes* to attach to the switch topology. The flows are then defined by the $n(n-1)$ source-destination pairs between each pair of endnodes. The intensity of the traffic can be controlled by the number of endnodes we attach to the original topology.

Different algorithms may be used to attach endnodes to the given topology. For example, concentration of endnodes attached to strategic parts of the network can artificially create congestion points (or *hot spots*) for study. By default, we

[7] As an easy way to prevent loop-freedom, we are only considering equal-distance alternative paths.

[8] Without the assumption of using routing tables that force all sources to the same destination to share the same partial routes, and the constraint of only considering equal-distance shortest path routes.

apply an algorithm that attaches endnodes to a switch with the smallest degree first. In physical terms, a switch with the smallest degree corresponds to one with the most ports left (assuming they all have the same number of ports).

Now the network capacity optimization problem can be stated as the problem of finding the best way to use the alternative routes so as to maximize the throughput of the given flows.

We build a heuristic search procedure to tackle this problem since we know finding the optimal solution will be very time consuming. We define the *usage* level, $U(i)$, of a link (i) to be the number of flows traversing it. Since we are using only equal-cost paths, the average link usage for all L links is constant:

$$\overline{U} = \sum_{1}^{L} U(i) = n(n-1)\overline{D}/L$$

This is because no matter which alternative paths we use, the average link usage can be derived from the average path distance $\overline{D}$, as shown above.

However, as we try different alternative paths, the variance of the link usage changes. Intuitively, the smaller the variance of U, the higher the simultaneous traffic the switching network is able to carry. Therefore, for our heuristic search algorithm we use the variance of U as the criterion to try to minimize.

6.3. Fault Tolerance

Fault tolerance is easier to define. For each source and destination, the goal is to have an alternative path available that is as different from the optimal path as possible. The fault tolerance, $F(i,j)$ between endnodes i and j is thus defined as the percentage of difference between the optimal path and the most differing alternative path. The fault tolerance of the network is thus the average fault tolerance over all the endnode pairs.

6.4. Network Optimization Algorithms

The optimization algorithm for maximizing capacity has three steps:

1. Remove duplicate links between nodes, and compute equal-distance alternative paths at the time the routing table is built;

2. Add duplicate links back, and load-balance flows across the duplicate links;

3. Consider other alternative paths using a hill-climbing algorithm.

The reason duplicate links are treated separately is that we find it very effective (computation time-wise) to single them out in maximizing the capacity. The idea is very simple. When you add a duplicate link to your network, suddenly all the flows going through the original link can be load-balanced. Since the routing table does not depend on the source of the flow, the balance of load between the duplicate links may not be perfect.

More specifically, let the added duplicate link be $i \rightarrow j$, and let us define f_{ik} as the number of flows passing through link $i \rightarrow j$ via i. We sort f_i (the vector of f_{ik}) in descending order. Then we iteratively assign the elements of f_i to the duplicate link that has the smaller amount of traffic.

In step 3, we consider the rest of the alternative routes. This is somewhat more complicated. We put all the possible choices of alternative routes in a combined list, and randomly (or use some other criterion to) pick one; swap the alternative with the current optimal path; check for improvement. We accept the alternative only if there is an improvement. After T trials without further improvements, the algorithm stops. The value of T is set to a percentage of the number of potential alternative paths that can be tried. Obviously, a large value of T will give potentially a more optimal result, but take a long time, and vice versa.

The same hill-climbing technique can be applied to optimize fault tolerance. Since we cannot optimize for both criteria at the same time, we chose to optimize the capacity and measure the resulting fault tolerance of the network.[9]

6.5. Comparison

For each of the protocols (OPT, UD and BFOT) we used the hill-climbing algorithm to minimize the standard deviations [10]. Protocol OPT is consistently better then UD and BFOT. Protocol OPT does not have the deadlock-free constraint, hence there are more alternative routes, so OPT is expected to perform better. Our interest is really in how BFOT and UD compared. Out of the 46 topologies, BFOT out-performed UD 35 to 11 times.

The fact that BFOT performs better than UD is very encouraging. Our speculation is that since UD is based on a tree, for many topologies the root becomes a natural concentration point for traffic. The ordered-tree approach, on the other hand, can better avoid such a single point of concentration. As we discussed earlier, however, the performance of the ordered-tree algorithm is quite sensitive to how the partition is done. This is definitely an area worthy of further investigation.

[9] If a given level of fault tolerance is required, then it is also possible to optimize network capacity using the given fault tolerance level as a constraint.

[10] Recall that a smaller standard deviation indicates better *load-balancing*, hence potentially higher achievable throughput.

The fault-tolerance metric measures the percentage of the time in which a link is removed that an alternative route is readily available. In this case, OPT again consistently perform better as expected. This time protocol UD beat BOPT 38 times out of 46.

7. Concluding Remarks

7.1. Summary

There are several results in this paper:

1. First, we introduced a new method of computing deadlock-free routing tables, called the Ordered-tree algorithm, and proved its correctness.
2. Second, we described two implementations of the new algorithm. One (APOT) gives the best result but takes longer to compute, while the other (BFOT) gives almost optimal results but is much faster and the computation can be distributed.
3. Third, we developed several metrics for evaluating the different algorithms: average path length, time of computation, network capacity, and fault-tolerance.
4. And finally, we compared the algorithms using these metrics.

We believe the Ordered-tree algorithm demonstrates great potential, especially considering the metric of network capacity.

7.2. Other Observations and Future Work

We have a third implementation of the ordered tree algorithm (in addition to APOT and BFOT), based on Fibonacci heaps. This implementation produced a dramatic improvement over APOT in terms of time of computation. Unlike BFOT, it still computes optimal routes. In comparison to BFOT, its computation time is still significantly higher (an order of magnitude for the topologies we tried).

As we alluded to several times in the paper, an interesting direction for future work is to investigate how to better partition the network in the ordered-tree algorithm. In other words, given a topology, how to evaluate which links belong to lower layers. We have tried some methods in prioritizing the links, but the results were not consistent across different topologies. But the results show that sorting the links can make a significant impact and is worthy of further studies.

Another important aspect not addressed in this paper is how to compute incremental updates to the routing information for small changes to the topology. In such situations, it is desirable to avoid recomputation of the whole routing table and provide speedy updates to the switches that need them.

8. Acknowledgements

We are grateful to Jose Duato for his help in introducing us to the literature on this topic, and Timothy Pinkston and Wai Hong Ho for providing us IRFlexSim and support. We have enjoyed interesting discussions with Lev Markov about alternative approaches to this problem. We also thank Bob Sproull for reading the manuscripts and giving us valuable comments.

References

[1] http://www.usc.edu/dept/ceng/pinkston/tools.html.

[2] I. T. Association. Infiniband(tm) architecture specification. *http://www.infinibandta.com.*

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw Hill, 1996.

[4] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 1987.

[5] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* 1997.

[6] M. Schroeder et al. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications, Vol. 9, No. 9, October*, 1991.

[7] A. Robles J.C. Sancho and J. Duato. Effective strategy to compute forwarding tables for infiniband networks. *ICPP01*, 2001.

[8] S. S. Owicki and A. R. Karlin. Factors in the performance of the an1 computer network. Technical report, Systems Research Center, DEC, Research Report 88, 1992.

[9] J. C. Sancho and A. Robles. Improving the up*/down* routing scheme for networks of workstations. *CANPC00*, 2000.

[10] F. Silla and J. Duato. High-performance routing in networks of workstations with irregular topology. *TPDS*, 2000.