Storage-Efficient, Deadlock-Free Packet Routing Algorithms for Torus Networks*

Robert Cypher[†] Luis Gravano[‡]

Abstract

We present two new packet routing algorithms for parallel computers with torus interconnection networks of arbitrary size and dimension. Both algorithms use only minimal length paths, are fully adaptive in the sense that all minimal length paths may be used to avoid congestion, and are free of deadlock, livelock and starvation. Algorithm 1 requires only three central queues per routing node. It is the first known minimal length packet routing algorithm for torus networks which requires a constant number of queues per node, regardless of the size and dimension of the torus. In fact, the requirement of three queues per node is optimal, as no such algorithm is possible when all nodes have two or fewer queues. Algorithm 2 requires only that each node have two input buffers per edge. It is the first known minimal-fully-adaptive packet routing algorithm for torus networks which does not require central queues and which does not require any node to have more than two input or two output buffers per edge. Both algorithms are simple and appear to be well-suited to VLSI implementation. They can be used with either store-and-forward or virtual cut-through routing.

*Based in part on "Adaptive, Deadlock-Free Packet Routing in Torus Networks with Minimal Storage", by R. Cypher and L. Gravano, which appeared in *Proceedings of the 1992 International Conference on Parallel Processing*, Aug. 17–21, 1992, vol. III, pp. 204–211

[†]IBM Almaden Research Center, San Jose, CA 95120

[‡]Computer Science Department, Stanford University, Stanford, CA 94305. This work was performed while the author was at CRAAG, IBM Argentina, 1300 Buenos Aires, Argentina.

1 Introduction

Parallel computers with mesh and torus interconnection networks are able to support many scientific and image processing applications very efficiently. In addition, d-dimensional meshes and tori can be laid out in d dimensions using only short wires, and they can be built using identical boards, each of which requires only a small number of pins for connections to other boards. Because of these advantages, a large number of distributed memory parallel computers utilize a mesh or torus interconnection network. Examples include the MPP from Goodyear Aerospace [1], the MP-1 from MasPar [23], the Paragon from Intel Scientific, and the J-Machine from MIT [24].

The primary disadvantages of mesh and torus interconnection networks are their relatively large diameters and relatively small bisection bandwidths [27], both of which limit their ability to support global communication. However, note that given d-dimensional mesh and torus computers of equal size, the torus computer has approximately half the diameter and twice the bisection bandwidth of the mesh computer. Furthermore, torus networks are node symmetric, so all nodes in a torus are identical and no region of the torus is particularly likely to suffer from congestion. In contrast, mesh networks are not node symmetric, and their lack of symmetry can cause certain regions of the mesh to suffer from congestion [20]. As a result, torus networks could play an important role in the next generation of parallel computers.

The performance of a parallel computer is largely dependent on the performance of its communication network, so a great deal of research has been devoted to developing efficient routing algorithms. A fundamental requirement of any routing algorithm is that it guarantee that all messages will eventually be delivered to their destinations. More specifically, it is essential that the routing algorithm be free of deadlock, livelock, and starvation. Also, in order to obtain good performance it is desirable that the routing algorithm use a shortest possible route for each message. Such a routing algorithm is said to be *minimal*. Another key property of a routing algorithm is whether or not it allows messages to adapt to traffic conditions by selecting alternate routes. A minimal routing algorithm which allows each message to take all of its shortest routes is said to be *minimal-fully-adaptive*.

Routing algorithms can be further classified by the type of switching mode that they utilize. In *store-and-forward* routing [22] each packet is stored completely in a node before being sent to the next node in its path. In contrast, *wormhole* routing [7] breaks each packet into small pieces called *flits*. As soon as a flit has been received by a node, it is sent to the next node in its path without waiting for the remaining flits to arrive. This creates a "worm" of flits which follow one another through the network. If the head of this worm encounters congestion, the entire worm is prevented from making progress. Finally, *virtual cut-through* routing [17] is similar to wormhole routing in that each packet is sent as a worm of flits which follow one another through the network. However, virtual cut-through routing buffers the entire worm in the node in which congestion occurs.

In general, store-and-forward routing is a simple technique which works well when the packets are small in comparison with the channel widths. Both wormhole routing and virtual cut-through routing perform well with long messages, assuming they encounter little or no congestion. Under heavy traffic conditions, however, virtual cut-through routing performs significantly better than wormhole routing due to the fact that blocked packets are stored in only one node. The primary disadvantage of virtual cut-through routing is that it requires significantly more storage than wormhole routing.

Many techniques have been developed to reduce the storage requirements of deadlock-free store-and-forward and virtual cut-through routing algorithms. These techniques can be divided into two classes, namely those which require only central queues [10, 12, 16, 18, 22, 25, 28, 29] and those which require that each node have queues (often called buffers) that are associated with each edge that is incident to the node [9, 15, 19]. Algorithms in the first class often require less storage than those in the second class. However, the central queues can become sequential bottlenecks, so algorithms in the second class may offer better performance. In addition, it should be noted that deadlock-free wormhole routing algorithms [3, 5, 6, 7, 8, 11, 2, 13, 21, 26] can be used to obtain deadlock-free store-and-forward and virtual cut-through algorithms in the second class.

No previously published technique yields a minimal store-and-forward or virtual cut-through routing algorithm that requires only a constant number of queues per node in tori of arbitrary size and dimension. For example, the "hops-so-far" scheme [22] requires that each node have more queues than the diameter of the torus. The routing algorithm described by Jesshope, Miller and Yantchev [16] and by Yantchev and Jesshope [29] is minimal-fully-adaptive, but it requires eight queues per node in a two-dimensional torus and more queues per node in higher dimensional tori. The techniques created by Pifarré, Gravano, Felperin and Sanz [25] and by Felperin, Laffitte, Buranits and Sanz [9] lead to minimal-fully-adaptive routing algorithms that require $2d$ queues per node in d-dimensional tori, where $d \geq 2$. In contrast, we will present a minimal-fully-adaptive routing algorithm which is free of deadlock, livelock, and starvation and yet requires only three central queues per node, regardless of the size and dimension of the torus. Furthermore, the requirement of three queues per node is optimal, as it has been proven that it is impossible to create a minimal deadlock-free routing algorithm for tori (whether or not it is adaptive) which uses only two queues per node [4].

We will also present a minimal-fully-adaptive routing algorithm for torus networks of arbitrary size and dimension which is free of deadlock, livelock, and starvation and yet requires only that each node have two input buffers per edge. All previously known algorithms with the same properties have either required central queues or have required that some node have at least three input or three output buffers associated with some edge [2]. Both of the algorithms presented here can operate with either store-and-forward or virtual cut-through switching. In addition, they are quite simple and thus appear to be well-suited to VLSI implementation. These algorithms use the same principles as those used by Günther [14] and by Pifarré *et al.* [25] for the design of deadlock-free packet routing algorithms.

The remainder of this paper is organized as follows. The formal routing model and some definitions are given in Section 2. Routing Algorithms 1 and 2 are presented in Sections 3 and 4, respectively. Some extensions to both of the routing algorithms are discussed in Section 5 and conclusions are given in Section 6.

2 Preliminaries

2.1 Torus Networks

Both of the routing algorithms presented in this paper operate on an $k_{d-1} \times k_{d-2} \times \ldots \times k_0$ d-dimensional torus network, where $d \geq 1$ and $k_i \geq 2$ for all i, $0 \leq i < d$. The torus network contains $n = \prod_{i=0}^{d-1} k_i$ nodes. Each node in the torus has a unique label of the form $(a_{d-1}, a_{d-2}, \ldots, a_0)$, where $0 \leq a_i < k_i$ for all i, $0 \leq i < d$. Each node $(a_{d-1} \ldots a_{i+1}, a_i, a_{i-1} \ldots a_0)$ is connected to all of those nodes of the form $(a_{d-1} \ldots a_{i+1}, a_i \pm 1 \bmod k_i, a_{i-1} \ldots a_0)$, where $0 \leq i < d$. The parameter i will be referred to as the *dimension* of the edge. Note that the class of $k_{d-1} \times k_{d-2} \times \ldots \times k_0$ d-dimensional torus networks contains the class of k-ary d-cubes as a subset.

Edges that connect nodes of the form $(a_{d-1} \ldots a_{i+1}, k_i - 1, a_{i-1} \ldots a_0)$ and $(a_{d-1} \ldots a_{i+1}, 0, a_{i-1} \ldots a_0)$ are called *wraparound* edges, and all other edges are called *internal* edges (see Figure 1). All of the edges are bidirectional. A transfer of a packet from a node of the form $(a_{d-1} \ldots a_{i+1}, a_i, a_{i-1} \ldots a_0)$ to a node of the form $(a_{d-1} \ldots a_{i+1}, a_i + 1 \bmod k_i, a_{i-1} \ldots a_0)$ will be said to occur in the *positive direction*, and transfers in the opposite direction will be said to occur in the *negative direction* (see Figure 2).

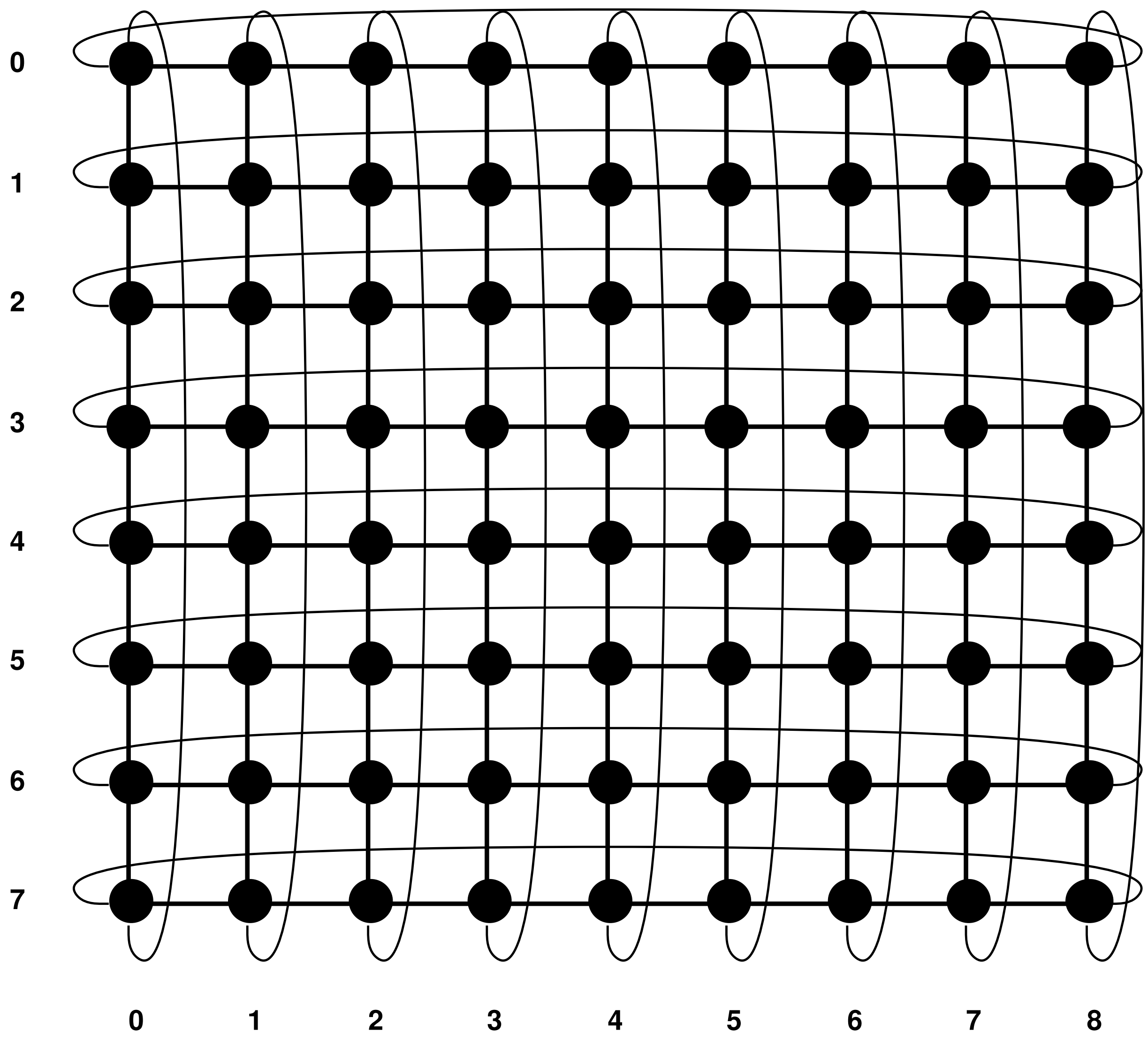

Figure 1: An 8×9 torus with internal edges highlighted.

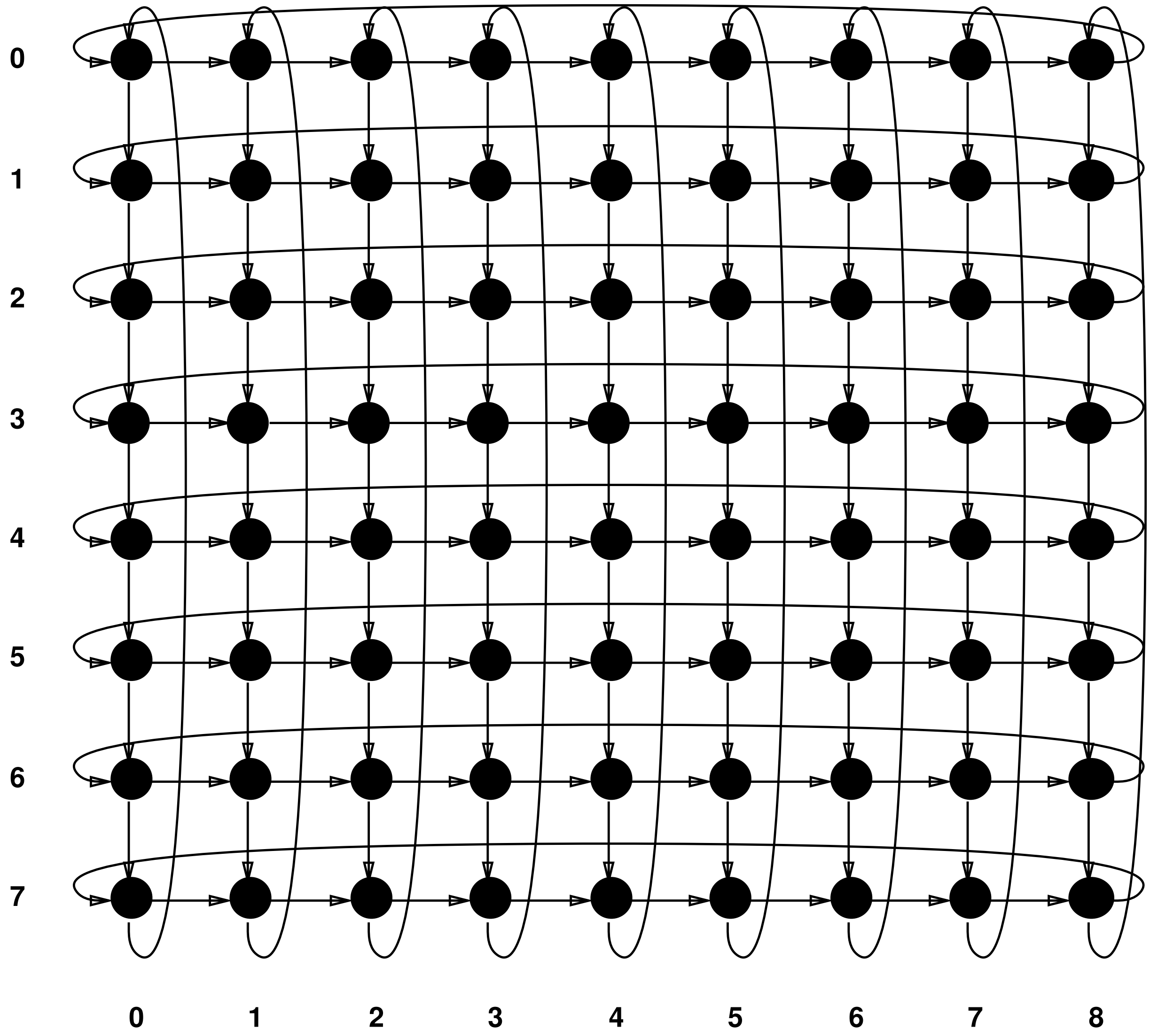

Figure 2: An 8×9 torus with positive direction edges.

2.2 The Routing Model

We will now formally define the model used by both of the routing algorithms presented in this paper. For the sake of simplicity, we will assume throughout this section and Sections 3 and 4 that store-and-forward routing is used and that each of the queues has sufficient storage for exactly one packet. These assumptions will be relaxed in Section 5.

Each node in the torus network contains a set of queues, one of which will be called an *injection queue*, another one of which will be called a *delivery queue*, and the remainder of which will be called *standard queues*. Packets can enter the torus network only by being placed in an empty injection queue in their source node, and they can be removed from the network only when they are in the delivery queue of their destination node. It is important to realize that the injection and delivery queues are introduced only to simplify the description of the model, and that they need not be present in an actual network. As a result, we will include only standard queues when we count the number of queues per node that are required by a routing algorithm.

If each of the standard queues in a node can be directly accessed from all of the node's input ports, the standard queues will be called *central queues*. On the other hand, if each standard queue can only be directly accessed from one of the node's input ports, the standard queues will be called *edge queues*.

An adaptive routing algorithm specifies which movements between queues are allowed. More precisely, given the source and destination nodes of a packet and given the queue in which the packet is currently stored, an adaptive routing algorithm specifies a set of queues to which the packet may be moved. We will call this set of queues to which the packet may be moved the packet's *waiting set*. All of the queues in a waiting set must either be in the node which currently holds the packet or in neighboring nodes (that is, nodes that are connected by an edge to the node currently holding the packet). Injection queues are never allowed to appear in waiting sets, and the waiting set of a packet that is in a delivery queue must be empty. Thus injection and delivery queues are used only for placing new packets in the network and for removing packets once they have reached their destination. A new packet with an arbitrary destination may be placed in an empty injection queue at any time. The routing network moves a packet from one queue to another only if the latter queue is in the packet's waiting set. When a packet is moved from one queue to another, it occupies both of the queues for a finite amount of time, after which the former queue becomes empty.

We will also assume that the torus network has the following properties:

P1: A packet that is in the delivery queue of its destination node will be removed from the network within finite time.

P2: No packet remains in a queue forever if there is a queue in its waiting set which is or becomes permanently empty.

P3: No packet remains in a queue forever while an infinite number of other packets enter and leave some queue in its waiting set.

Properties P1 and P2 are finite progress assumptions which state that packets never wait infinitely long for no reason. Property P3 is a fairness assumption which prevents the possibility of starvation once a packet has entered the network.

The following lemma will be essential in proving the desired properties of our torus routing algorithms. The proof is immediate from a proposition proven by Günther [14].

Lemma 2.1 (Günther) *Given a routing network that satisfies Properties P1, P2 and P3 above, given a total ordering of the queues in the network, and given an adaptive routing algorithm such that each packet either is in the delivery queue of its destination node or has a waiting set that contains a higher ordered queue than the one which it occupies, no packet will remain in a single queue forever.*

It is important to notice that Lemma 2.1 does not force the packets to be routed through the queues in ascending order. Instead, it merely requires that each packet always has the opportunity to move to a higher ordered queue, whether or not the queue to which it actually chooses to move is higher ordered. This distinction is critical, and it is the basis for the storage-efficiency of our torus routing algorithms.

Lemma 2.1 shows that when a routing network and a routing algorithm satisfy certain properties, every packet will continue to move. Thus we will use Lemma 2.1 to prove freedom from deadlock and to prove that once a packet has been injected into the network, it will not suffer from starvation. However, Lemma 2.1 does not address freedom from livelock (in which a packet circles infinitely through the network) or freedom from starvation at the time of injection (in which a packet is never able to be injected into the network because the injection queue remains occupied). Thus, different techniques will be required to prove freedom from livelock and freedom from starvation at the time of injection.

2.3 Useful Node Orderings

We will now define four different total orderings of the nodes in a torus network. These orderings, which will be used throughout the paper, will be helpful both in defining the routing algorithms and in proving that they are free of deadlock, livelock and starvation. In particular, these node orderings break the symmetry of the torus network and they will provide a basis for the queue orderings that will be defined later in the paper.

The first ordering, which is called the *right-increasing* ordering, is simply a standard row-major ordering of the nodes (see Table 1). The second ordering, which is called the *left-increasing* ordering, is simply the reverse of the right-increasing ordering (see Table 2). The third ordering, which is called the *inside-increasing* ordering, assigns the smallest values to nodes near the wraparound edges of the torus and the largest values to nodes near the center of the torus (see Table 3). The fourth ordering, which is called the *outside-increasing* ordering, is simply the reverse of the inside-increasing ordering (see Table 4).

More formal definitions of these orderings are as follows. Given any integer i, $0 \leq i < d$, let

$$g(i) = \prod_{j=0}^{i-1} k_j$$

(thus $g(0) = 1$). Given any torus node label of the form $(a_{d-1}, a_{d-2}, \ldots, a_0)$, let

$$Eval((a_{d-1}, a_{d-2}, \ldots, a_0)) = \sum_{i=0}^{d-1} g(i) a_i.$$

Thus the function $Eval$ assigns a unique integer in the range 0 through $n-1$ to each of the nodes. Note that $Eval$ can be viewed as interpreting a node label as a mixed radix representation of an integer.

The four total orderings of the torus nodes will be obtained by first relabeling the nodes according to some function and then evaluating the new labels as integers. Given any integer $k_i \geq 2$ and any integer a_i where $0 \leq a_i < k_i$, let

$$f_R(a_i, k_i) = a_i,$$

let

$$f_L(a_i, k_i) = k_i - a_i - 1,$$

let

$$f_I(a_i, k_i) = \begin{cases} a_i & \text{if } a_i < \lfloor k_i/2 \rfloor \\ \lfloor 3k_i/2 \rfloor - a_i - 1 & \text{otherwise} \end{cases},$$

and let

$$f_O(a_i, k_i) = \begin{cases} k_i - a_i - 1 & \text{if } a_i < \lfloor k_i/2 \rfloor \\ a_i - \lfloor k_i/2 \rfloor & \text{otherwise} \end{cases}.$$

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71

Table 1: The right-increasing ordering of the nodes in an 8×9 torus.

71	70	69	68	67	66	65	64	63
62	61	60	59	58	57	56	55	54
53	52	51	50	49	48	47	46	45
44	43	42	41	40	39	38	37	36
35	34	33	32	31	30	29	28	27
26	25	24	23	22	21	20	19	18
17	16	15	14	13	12	11	10	9
8	7	6	5	4	3	2	1	0

Table 2: The left-increasing ordering of the nodes in an 8×9 torus.

0	1	2	3	8	7	6	5	4
9	10	11	12	17	16	15	14	13
18	19	20	21	26	25	24	23	22
27	28	29	30	35	34	33	32	31
63	64	65	66	71	70	69	68	67
54	55	56	57	62	61	60	59	58
45	46	47	48	53	52	51	50	49
36	37	38	39	44	43	42	41	40

Table 3: The inside-increasing ordering of the nodes in an 8×9 torus.

71	70	69	68	63	64	65	66	67
62	61	60	59	54	55	56	57	58
53	52	51	50	45	46	47	48	49
44	43	42	41	36	37	38	39	40
8	7	6	5	0	1	2	3	4
17	16	15	14	9	10	11	12	13
26	25	24	23	18	19	20	21	22
35	34	33	32	27	28	29	30	31

Table 4: The outside-increasing ordering of the nodes in an 8×9 torus.

a_i:	0	1	2	3	4	5	6	7	8
$f_R(a_i, 9)$:	0	1	2	3	4	5	6	7	8
$f_L(a_i, 9)$:	8	7	6	5	4	3	2	1	0
$f_I(a_i, 9)$:	0	1	2	3	8	7	6	5	4
$f_O(a_i, 9)$:	8	7	6	5	0	1	2	3	4

Table 5: The functions $f_R(a_i, k_i)$, $f_L(a_i, k_i)$, $f_I(a_i, k_i)$ and $f_O(a_i, k_i)$ when $k_i = 9$.

Thus the function f_R orders the numbers 0 through $k_i - 1$ in increasing order from left to right, the function f_L orders them from right to left, the function f_I orders them from the outside to the inside, and the function f_O orders them from the inside to the outside. Examples of these four functions are shown in Table 5.

Given any torus node $(a_{d-1}, a_{d-2}, \ldots, a_0)$, let

$$Right((a_{d-1}, a_{d-2}, \ldots, a_0)) = Eval((f_R(a_{d-1}, k_{d-1}), f_R(a_{d-2}, k_{d-2}), \ldots, f_R(a_0, k_0))),$$

let

$$Left((a_{d-1}, a_{d-2}, \ldots, a_0)) = Eval((f_L(a_{d-1}, k_{d-1}), f_L(a_{d-2}, k_{d-2}), \ldots, f_L(a_0, k_0))),$$

let

$$Inside((a_{d-1}, a_{d-2}, \ldots, a_0)) = Eval((f_I(a_{d-1}, k_{d-1}), f_I(a_{d-2}, k_{d-2}), \ldots, f_I(a_0, k_0))),$$

and let

$$Outside((a_{d-1}, a_{d-2}, \ldots, a_0)) = Eval((f_O(a_{d-1}, k_{d-1}), f_O(a_{d-2}, k_{d-2}), \ldots, f_O(a_0, k_0))).$$

These orderings of the torus nodes will be referred to as *right-increasing*, *left-increasing*, *inside-increasing* and *outside-increasing*, respectively. A transfer of a packet from a node a to an adjacent node b will said to occur to the *right* (similarly, *left*, *inside* or *outside*) if and only if node a is smaller than node b when they are numbered in right-increasing (similarly, left-increasing, inside-increasing or outside-increasing) order (see Figures 3 and 4). For example, a transfer in the positive direction along an internal edge occurs to the right while a transfer in the positive direction along a wraparound edge occurs to the left. Also, note that negative direction transfers along wraparound edges occur to the right and inside while positive direction transfers along wraparound edges occur to the left and outside.

2.4 Notation

Let p be an arbitrary packet that is being routed in a torus network by either Algorithm 1 or Algorithm 2. The following notation will be used throughout the paper.

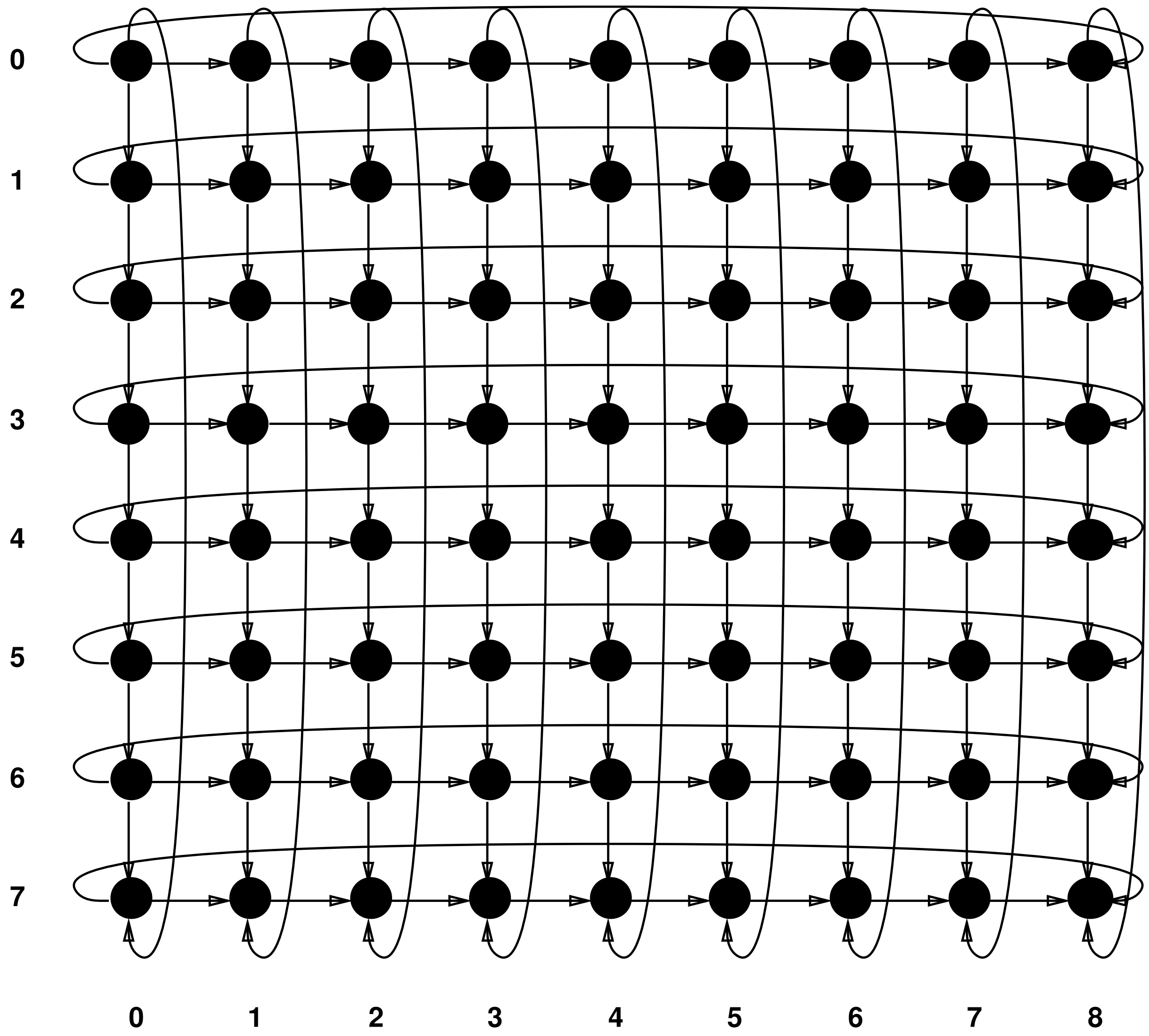

Figure 3: An 8×9 torus with right-increasing direction edges.

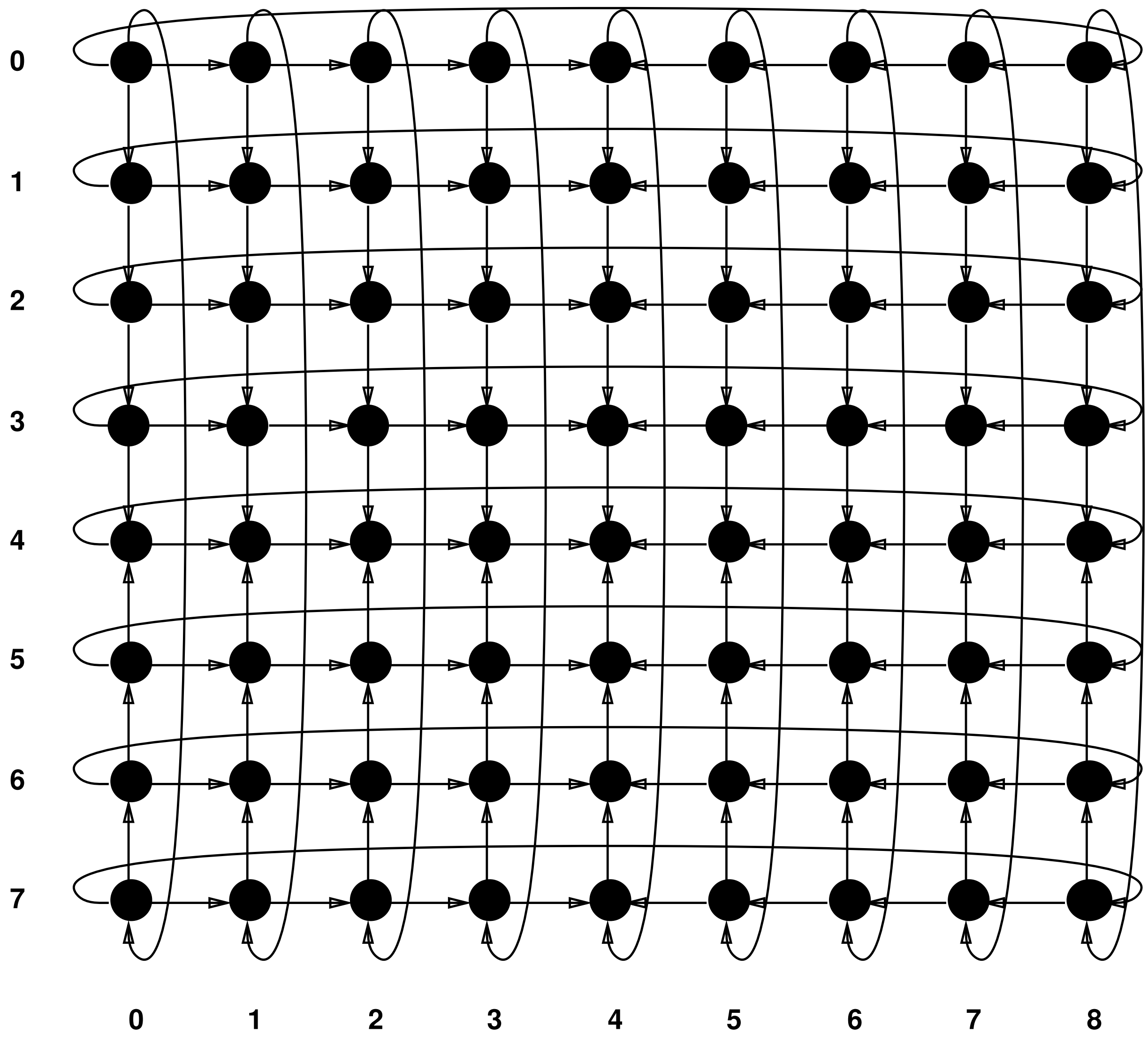

Figure 4: An 8×9 torus with inside-increasing direction edges.

$queue(p)$ denotes the queue in which p is currently stored.

$node(p)$ denotes the node in which p is currently located.

$source(p)$ denotes p's source node.

$dest(p)$ denotes p's destination node.

$wait(p)$ denotes p's waiting set (i.e. the set of queues to which the packet may be moved next).

$neighb(p)$ denotes the set of nodes that are torus neighbors of $node(p)$.

$ok\_nodes(p)$ denotes the subset of $neighb(p)$ consisting of those neighboring nodes that lie along a minimal length path from $node(p)$ to $dest(p)$.

$ok\_queues(p)$ denotes the set of standard queues in $ok\_nodes(p)$ that are directly accessible from $node(p)$. Therefore, if central queues are being used $ok\_queues(p)$ consists of all of the standard queues in $ok\_nodes(p)$. On the other hand, if edge queues are being used $ok\_queues(p)$ consists only of those standard queues in $ok\_nodes(p)$ that are associated with edges that are incident to $node(p)$.

3 Routing Algorithm 1

The first torus routing algorithm that we will present requires three central queues per node. In this section we will describe the algorithm and prove that it is free of deadlock, livelock and starvation.

3.1 Description

Routing Algorithm 1 operates on a torus network with three standard queues per node, designated A, B and C. It routes each packet along any minimal length path from its source node to its destination node. A formal description of Algorithm 1 is given below.

Algorithm 1.

Let p be an arbitrary packet that is being routed by Algorithm 1, let $q = queue(p)$, and let $x = node(p)$. Algorithm 1 creates p's waiting set $wait(p)$ according to the following rules.

Case 1: q is an injection queue.

In this case, $wait(p)$ consists of the A queue in x.

Case 2: q is an A queue.

In this case, there are two subcases.

Case 2a: $\exists y \in ok\_nodes(p)$ such that $Right(x) < Right(y)$.

In this subcase, $wait(p)$ consists of all of the A queues in $ok\_queues(p)$.

Case 2b: $\nexists y \in ok\_nodes(p)$ such that $Right(x) < Right(y)$.

In this subcase, $wait(p)$ consists of the B queue in x.

Case 3: q is a B queue.

In this case, there are two subcases.

Case 3a: $\exists y \in ok\_nodes(p)$ such that $Left(x) < Left(y)$.

In this subcase, $wait(p)$ consists of all of the B queues in $ok\_queues(p)$.

Case 3b: $\nexists y \in ok\_nodes(p)$ such that $Left(x) < Left(y)$.

In this subcase, $wait(p)$ consists of the C queue in x.

Case 4: q is a C queue.

In this case, there are two subcases.

Case 4a: $x \neq dest(p)$.

In this subcase, $wait(p)$ consists of all of the C queues in $ok\_queues(p)$.

Case 4b: $x = dest(p)$.

In this subcase, $wait(p)$ consists of the delivery queue in x.

Case 5: q is a delivery queue.

In this case, $wait(p)$ is the empty set.

Informally, each packet moves from its injection queue to the A queue in the same node and it remains in A queues as long as it is possible for it to move to the right along at least one dimension. It then moves to the B queue in its current node and it remains in B queues as long as it is possible for it to move to the left along at least one dimension. It then moves to the C queue in its current node and it remains in C queues until it arrives at its destination node. It then moves to the delivery queue in its destination node and is eventually delivered.

For example, consider a packet p that is routed from source node $(4, 7)$ to destination node $(2, 2)$ in an 8×9 torus. Assume that p takes the route $(4, 7) \rightarrow (3, 7) \rightarrow (3, 8) \rightarrow (3, 0) \rightarrow (2, 0) \rightarrow (2, 1) \rightarrow (2, 2)$ (see Figure 5). The packet p will first be stored in the injection queue in node $(4, 7)$. At this point, Case 1 in the definition of Algorithm 1 applies and $wait(p)$ consists of the A queue in node $(4, 7)$. Once p moves to the A queue in node $(4, 7)$, $ok\_nodes(p) = \{(3, 7), (4, 8)\}$. Because $(4, 8) \in ok\_nodes(p)$ and $Right((4, 7)) < Right((4, 8))$, Case 2a in the definition of Algorithm 1 applies and $wait(p)$ consists of the A queues in $(3, 7)$ and $(4, 8)$.

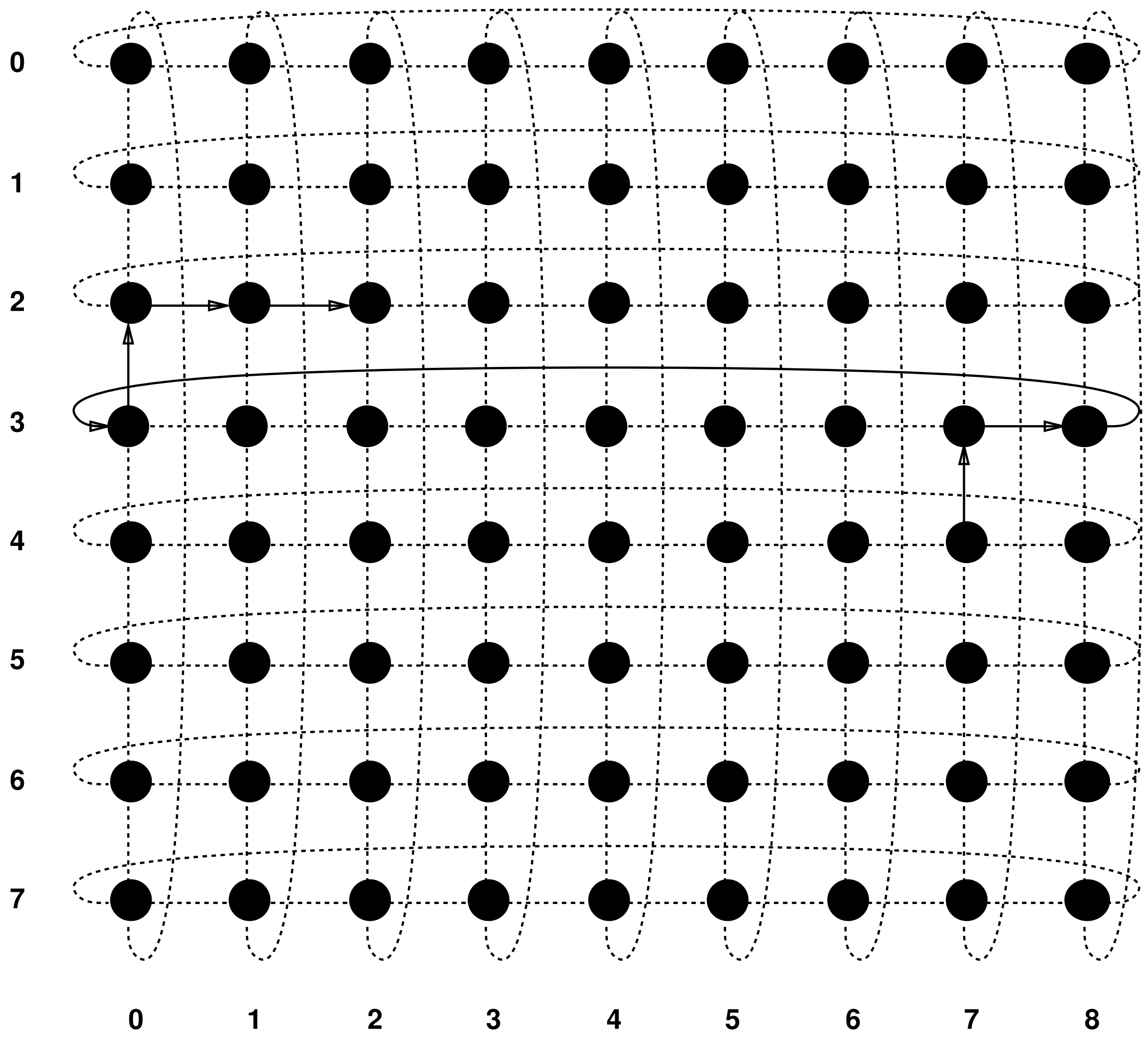

Figure 5: An example of a route used by Algorithm 1 in an 8×9 torus.

Overall, packet p will be stored in the injection queue in node $(4,7)$, the A queues in nodes $(4,7)$, $(3,7)$ and $(3,8)$, the B queues in nodes $(3,8)$, $(3,0)$ and $(2,0)$, the C queues in nodes $(2,0)$, $(2,1)$ and $(2,2)$, and the delivery queue in node $(2,2)$, from which it will be delivered.

3.2 Freedom from Deadlock, Livelock and Starvation

We will now prove that routing Algorithm 1 is free of deadlock, livelock and starvation. We will begin by proving several lemmas. The following lemma shows that packets that are stored in C queues only move to the inside. This fact will be crucial to our proof of freedom from deadlock.

Lemma 3.1 *Let p be any packet that is being routed by Algorithm 1 and let $(a_{d-1}, a_{d-2}, \ldots, a_0)$ denote the address of node(p). If queue(p) is a C queue, then for each dimension i, $0 \le i < d$, either p requires no further moves along dimension i or p's next move along dimension i will occur to the inside.*

Proof: First, note that the fact that the algorithm is minimal implies that p makes a total of at most $\lfloor k_i/2 \rfloor$ moves along dimension i. Also, note that all minimal length paths from *source*(p) to *dest*(p) contain either only positive direction moves along dimension i or only negative direction moves along dimension i. Furthermore, note that if *queue*(p) is a C queue, p must have already moved at least once along dimension i, and all further moves that p makes along dimension i will be in the same direction[1]. There are three cases to consider.

Case 1: p requires no further moves along dimension i.
In this case, the lemma holds trivially.

Case 2: p requires a positive move along dimension i.
Note that in this case, all of p's previous moves along dimension i were also in the positive direction. Also, note that p must have already visited a node with the value $k_i - 1$ in dimension i, because otherwise it could not have moved from an A queue to a B queue. Furthermore, p must have then moved along a wraparound edge in dimension i, because otherwise it could not have moved from a B queue to a C queue. Therefore, the fact that p makes at most $\lfloor k_i/2 \rfloor$ moves along dimension i implies that $a_i < \lfloor k_i/2 \rfloor$, so p's next move along dimension i will occur to the inside.

[1] Note that when k_i is even and p must make a total of exactly $k_i/2$ moves along dimension i, p can traverse dimension i in either the positive or negative direction. Therefore, the fact that p has already moved along dimension i, and has thus established its direction of movement along dimension i, is significant.

Case 3: p requires a negative move along dimension i.
Note that in this case, all of p's previous moves along dimension i were also in the negative direction. Also, note that p must have already visited a node with the value 0 in dimension i, because otherwise it could not have moved from a B queue to a C queue. We have two subcases to consider.

Case 3a: $a_i = 0$. In this subcase, p's next move along dimension i will be in the negative direction along a wraparound edge, which is a move to the inside.

Case 3b: $a_i \neq 0$. In this subcase, p must have already moved in the negative direction along a wraparound edge in dimension i. Therefore, the fact that p makes at most $\lfloor k_i/2 \rfloor$ moves along dimension i implies that $\lfloor k_i/2 \rfloor < a_i$, so p's next move will occur to the inside.

□

For the following lemma we will need to define a total ordering of the queues in the torus. This total ordering of the queues will be helpful in proving freedom from deadlock.

Definition. Let q be any queue in the torus network that is used by Algorithm 1, and let x denote the node in which q is located. Recall that n denotes the number of nodes in the torus network. The ranking function $Rank1(q)$ is defined as follows.

$$Rank1(q) = \begin{cases} Right(x) & \text{if } q \text{ is an injection queue} \\ n + Right(x) & \text{if } q \text{ is an } A \text{ queue} \\ 2n + Left(x) & \text{if } q \text{ is a } B \text{ queue} \\ 3n + Inside(x) & \text{if } q \text{ is a } C \text{ queue} \\ 4n + Right(x) & \text{if } q \text{ is a delivery queue} \end{cases}.$$

Lemma 3.2 *Let p be any packet that is being routed by Algorithm 1 and let $q = queue(p)$. Either q is the delivery queue in $dest(p)$ or there exists a queue $w \in wait(p)$ such that $Rank1(q) < Rank1(w)$.*

Proof: Let $x = node(p)$. We will consider each of the rules for the definition of $wait(p)$ separately.

Case 1: q is an injection queue.
In this case, let w be the A queue in x and note that $Rank1(q) < Rank1(w)$.

Case 2: q is an A queue.

In this case, there are two subcases.

Case 2a: $\exists y \in ok\_nodes(p)$ such that $Right(x) < Right(y)$.

In this subcase, let w be the A queue in y and note that $Rank1(q) < Rank1(w)$.

Case 2b: $\nexists y \in ok\_nodes(p)$ such that $Right(x) < Right(y)$.

In this subcase, let w be the B queue in x and note that $Rank1(q) < Rank1(w)$.

Case 3: q is a B queue.

In this case, there are two subcases.

Case 3a: $\exists y \in ok\_nodes(p)$ such that $Left(x) < Left(y)$.

In this subcase, let w be the B queue in y and note that $Rank1(q) < Rank1(w)$.

Case 3b: $\nexists y \in ok\_nodes(p)$ such that $Left(x) < Left(y)$.

In this subcase, let w be the C queue in x and note that $Rank1(q) < Rank1(w)$.

Case 4: q is a C queue.

In this case, there are two subcases.

Case 4a: $x \neq dest(p)$.

In this subcase, let y be any node in $ok\_nodes(p)$. It follows from Lemma 3.1 that $Inside(x) < Inside(y)$, so let w be the C queue in y and note that $Rank1(q) < Rank1(w)$.

Case 4b: $x = dest(p)$.

In this subcase, let w be the delivery queue in x and note that $Rank1(q) < Rank1(w)$.

Case 5: q is a delivery queue.

In this case, the lemma holds trivially.

□

The following lemma will allow us to prove that Algorithm 1 is free of livelock.

Lemma 3.3 *If p is any packet that is being routed by Algorithm 1, then p will be stored in at most a finite number of queues before being placed in the delivery queue of its destination node.*

Proof: Because p always takes a minimal length path to its destination, it visits only a finite number of nodes. Whenever p visits a node, it is stored in each of the five queues at most once each, because the routing algorithm visits the different types of queues in a monotonically increasing order. □

Combining the previous lemmas gives us the following theorem.

Theorem 3.4 *Routing Algorithm 1 is free of deadlock, livelock and starvation.*

Proof: From Lemmas 2.1 and 3.2, it follows that once a packet has been placed in an injection queue, it never remains in a single queue forever. Therefore, Algorithm 1 is free of deadlock and it is free of starvation for packets that have been injected into the network. From Lemma 3.3 and the fact that no packet remains in a single queue forever, it follows that every packet will reach the delivery queue of its destination node within finite time. Therefore, Algorithm 1 is free of livelock. Finally, the fact that no packet will remain in an injection queue forever implies that a full injection queue will always become empty again, so the algorithm is free of starvation for incoming packets. □

Thus routing Algorithm 1 is minimal-fully-adaptive and free of deadlock, livelock, and starvation, and yet it requires only three queues per node. The following theorem, which was proven by Cypher and Gravano [4], shows that the requirement of three queues per node is optimal, in the sense that no minimal algorithm which requires only two queues per node is free of deadlock.

Theorem 3.5 *Any torus routing algorithm of the type defined in Section 2.2 which is minimal and free of deadlock requires that some node have at least three queues.*

4 Routing Algorithm 2

The second torus routing algorithm that we will present uses edge queues rather than central queues. More specifically, Algorithm 2 requires that each node have two standard queues per each edge incident to the node. Each of these standard queues is used only by messages which are entering the given node along the associated edge. In the remainder of this section we will describe Algorithm 2 and prove that it is free of deadlock, livelock and starvation.

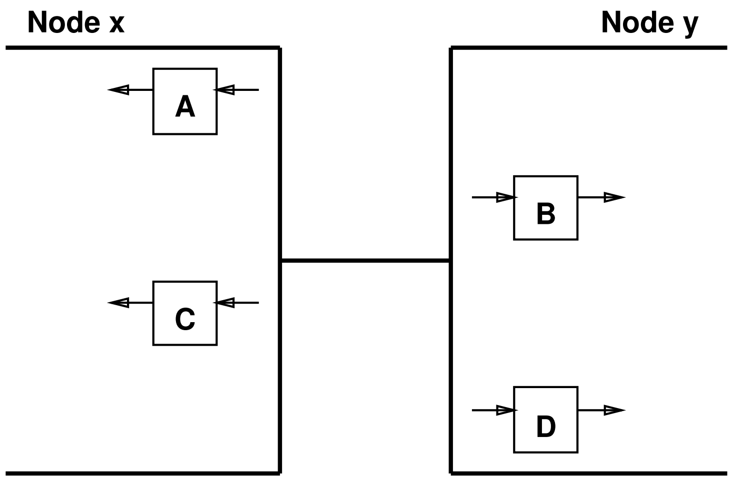

Figure 6: Queues associated with an edge connecting nodes x and y, where node x is smaller than node y when they are numbered in inside-increasing order.

4.1 Description

Each node in the torus network has two standard queues associated with each of the edges incident to the node. The standard queues are divided into four classes, designated A, B, C and D. The A and C queues are associated with edges along which packets have moved to the outside, while the B and D queues are associated with edges along which packets have moved to the inside. An example is shown in Figure 6.

Routing Algorithm 2 routes each packet along any minimal length path from its source node to its destination node. A formal description of Algorithm 2 is given below.

Algorithm 2.

Let p be an arbitrary packet that is being routed by Algorithm 2, let $q = queue(p)$, and let $x = node(p)$. Algorithm 2 creates p's waiting set $wait(p)$ according to the following rules.

Case 1: $x \neq dest(p)$ and q is an injection, A, or B queue.
In this case, there are two subcases.

Case 1a: $\exists y \in ok\_nodes(p)$ such that $Inside(x) < Inside(y)$.

In this subcase, $wait(p)$ consists of all of the A and B queues in $ok\_queues(p)$.

Case 1b: $\nexists y \in ok\_nodes(p)$ such that $Inside(x) < Inside(y)$.

In this subcase, $wait(p)$ consists of all of the C queues in $ok\_queues(p)$.

Case 2: $x \neq dest(p)$ and q is a C queue.
In this case, there are two subcases.

Case 2a: $\exists y \in ok\_nodes(p)$ such that $Outside(x) < Outside(y)$.

In this subcase, $wait(p)$ consists of all of the B and C queues in $ok\_queues(p)$.

Case 2b: $\nexists y \in ok\_nodes(p)$ such that $Outside(x) < Outside(y)$.

In this subcase, $wait(p)$ consists of all of the D queues in $ok\_queues(p)$.

Case 3: $x \neq dest(p)$ and q is a D queue.
In this case, $wait(p)$ consists of all of the D queues in $ok\_queues(p)$.

Case 4: $x = dest(p)$ and q is not a delivery queue.
In this case, $wait(p)$ consists of the delivery queue in x.

Case 5: $x = dest(p)$ and q is a delivery queue.
In this case, $wait(p)$ is the empty set.

For example, consider a packet p that is routed from source node $(5,7)$ to destination node $(3,2)$ in an 8×9 torus. Assume that p takes the route $(5,7) \rightarrow (4,7) \rightarrow (4,8) \rightarrow (4,0) \rightarrow (4,1) \rightarrow (3,1) \rightarrow (3,2)$ (see Figure 7). The packet p will first be stored in the injection queue in node $(5,7)$. At this point, $ok\_nodes(p) = \{(4,7),(5,8)\}$. Because $(4,7) \in ok\_nodes(p)$ and $Inside((5,7)) < Inside((4,7))$, Case 1a in the definition of Algorithm 2 applies and $wait(p)$ consists of the A and B queues in $(4,7)$ and $(5,8)$ that are accessible from $(5,7)$. Packet p will then be stored in a B queue in $(4,7)$, at which point $ok\_nodes(p) = \{(3,7),(4,8)\}$. Because $Inside((4,7)) > Inside((3,7))$ and $Inside((4,7)) > Inside((4,8))$, Case 1b applies and $wait(p)$ consists of the C queues in $(3,7)$ and $(4,8)$ that are accessible from $(4,7)$.

Overall, packet p will be stored in the injection queue in node $(5,7)$, a B queue in node $(4,7)$, C queues in nodes $(4,8)$ and $(4,0)$, a B queue in node $(4,1)$, an A queue in node $(3,1)$, a B queue in node $(3,2)$, and a delivery queue in node $(3,2)$, from which it will be delivered. Note that, unlike Algorithm 1, Algorithm 2 allows packets to move from C queues back to B queues and from B queues back to A queues. However, we will see that this cannot cause a problem, because once again the queues can be ordered so that each packet always has the possibility of moving to a higher ordered queue (or being delivered).

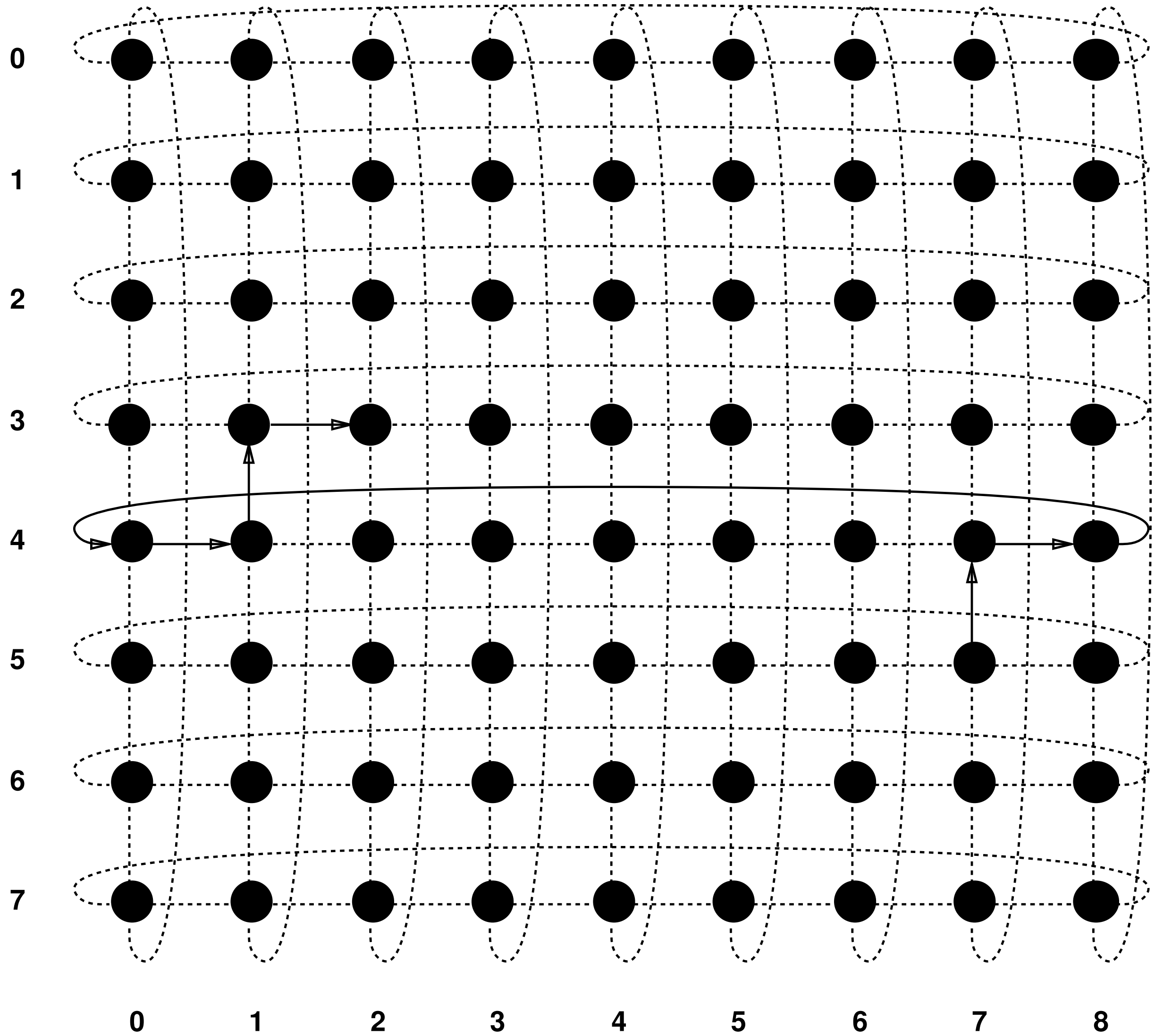

Figure 7: An example of a route used by Algorithm 2 in an 8×9 torus.

Finally, note that Case 3 in the definition of Algorithm 2 forces a packet that is in a D queue to remain in D queues until it reaches its destination. Because a D queue can only be entered when moving to the inside, this rule prevents packets in D queues from moving to the outside. Fortunately, Lemma 4.1 will prove that packets in D queues only need to move to the inside, so no adaptivity is lost.

4.2 Freedom from Deadlock, Livelock and Starvation

We will now prove that routing Algorithm 2 is free of deadlock, livelock and starvation. The proofs in this section are directly analogous to those in Section 3.2 and will not be repeated here.

Lemma 4.1 *Let p be any packet that is being routed by Algorithm 2 and let $(a_{d-1}, a_{d-2}, \ldots, a_0)$ denote the address of node(p). If queue(p) is a D queue, then for each dimension i, $0 \leq i < d$, either p requires no further moves along dimension i or p's next move along dimension i will occur to the inside.*

For the following lemma we will need to define a total ordering of the queues in the torus. This total ordering of the queues will be helpful in proving freedom from deadlock.

Definition. Let q be any queue in the torus network that is used by Algorithm 2, and let x denote the node in which q is located. Recall that n denotes the number of nodes in the torus network. The ranking function $Rank2(q)$ is defined as follows.

$$Rank2(q) = \begin{cases} Inside(x) & \text{if } q \text{ is an injection queue} \\ n + Inside(x) & \text{if } q \text{ is an } A \text{ queue} \\ 2n + Inside(x) & \text{if } q \text{ is a } B \text{ queue} \\ 3n + Outside(x) & \text{if } q \text{ is a } C \text{ queue} \\ 4n + Inside(x) & \text{if } q \text{ is a } D \text{ queue} \\ 5n + Inside(x) & \text{if } q \text{ is a delivery queue} \end{cases}.$$

Lemma 4.2 *Let p be any packet that is being routed by Algorithm 2 and let $q = queue(p)$. Either q is the delivery queue in dest(p) or there exists a queue $w \in wait(p)$ such that $Rank2(q) < Rank2(w)$.*

The following lemma will allow us to prove that Algorithm 2 is free of livelock.

Lemma 4.3 *If p is any packet that is being routed by Algorithm 2, then p will be stored in at most a finite number of queues before being placed in the delivery queue of its destination node.*

Combining the previous lemmas gives us the following theorem.

Theorem 4.4 *Routing Algorithm 2 is free of deadlock, livelock and starvation.*

5 Extensions

To this point we have focused on the simplest forms of our routing algorithms in order to prove some of their properties. In this section we will briefly mention some of the ways in which the routing algorithms can extended while maintaining their freedom from deadlock, livelock and starvation.

5.1 Larger Queues

One possible extension is to increase the size of the queues so that they can hold more than a single packet each. The routing algorithms will maintain their freedom from deadlock, livelock and starvation even if the queues operate in a strictly FIFO manner.

5.2 Common Queues

Another alternative would be to add a pool of "common queues" which could be used by any messages, in addition to the standard queues required by the algorithms in order to prevent deadlock. This approach is basically the same as the common buffer pool technique introduced by Merlin and Schweitzer [22]. Note that the use of common queues allows for more efficient use of the storage in the nodes than does the use of larger queues, as some of the storage is available to all of the packets attempting to enter the node. Because both common queues and larger queues require more storage than is required by the basic algorithms, it might appear that the goal of reducing the storage required for preventing deadlock is not relevant when these techniques can be used. However, reducing the amount of storage that is reserved for deadlock prevention could improve the utilization of the storage even when additional storage space is available.

5.3 Virtual Cut-Through

Finally, virtual cut-through routing can be used in place of store-and-forward routing. The implementation of virtual cut-through routing should exhibit the same deadlock properties as were assumed for the store-and-forward routing model that was presented. In particular, the virtual cut-through implementation should not allow packets to progress to a new queue unless it is guaranteed that sufficient storage for the entire packet will eventually be available at that queue.

6 Conclusions

We have presented two new packet routing algorithms for torus networks of arbitrary size and dimension. Both algorithms are minimal-fully-adaptive and are free of deadlock, livelock and starvation. Algorithm 1's storage requirements are smaller than those of any previously known algorithm with the same properties, and it is the first such algorithm to require a constant number of queues per node, regardless of the size and dimension of the torus. Furthermore, Algorithm 1's storage requirements are provably optimal, in that any minimal, deadlock-free algorithm for tori based on the model presented in Section 2.2 must have at least 3 queues in some node [4]. Algorithm 2 is the first minimal-fully-adaptive deadlock-free algorithm known for torus networks which requires only that each node have two input buffers per edge.

The torus has several significant advantages over the mesh. However, the presence of cycles in each dimension of the torus has made the development of deadlock-free routing algorithms more difficult than for the mesh. It is hoped that these routing algorithms will help in the development of efficient parallel computers based on torus interconnection networks.

References

[1] Kenneth E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, 29(9):836–840, September 1980.

[2] P. Berman, L. Gravano, G. D. Pifarré, and J.L.C. Sanz. Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks. In *Proceedings of the 4th Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1992.

[3] G.-M. Chiu, S. Chalasani, and C. S. Raghavendra. Flexible, fault-tolerant routing criteria for circuit-switched hypercubes. In *Proc.* 11^{th}

International Conference on Distributed Computing Systems. IEEE, 1991.

[4] Robert Cypher and Luis Gravano. Requirements for deadlock-free adaptive packet routing. In *Proc. 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 25–33, 1992.

[5] W. J. Dally. Fine-grain message passing concurrent computers. In *Proc. 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 2–12, 1988.

[6] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4:466–475, April 1993.

[7] W. J. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36:547–553, May 1987.

[8] J. Duato. Deadlock-free adaptive routing algorithms for multicomputers: Evaluation of a new algorithm. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 840–847, 1991.

[9] Sergio A. Felperin, Hernán Laffitte, Guillermo Buranits, and Jorge L.C. Sanz. Deadlock-free minimal packet routing in the torus network. Technical Report TR:91-22, IBM Argentina, CRAAG, 1991.

[10] B. Gavish, P.M. Merlin, and P.J. Schweitzer. Minimal buffer requirements for avoiding store-and-forward deadlock. Technical Report RC 6672, IBM T.J. Watson Research Center, August 1977.

[11] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *Proc. 19th Intl. Symposium on Computer Architecture*, pages 278–287, 1992.

[12] Inder S. Gopal. Prevention of store-and-forward deadlock in computer networks. *IEEE Transactions on Communications*, 33(12):1258–1264, December 1985.

[13] L. Gravano, G. D. Pifarré, G. Denicolay, and J.L.C. Sanz. Adaptive deadlock-free worm-hole routing in hypercubes. In *Proceedings of the 6th International Parallel Processing Symposium*. IEEE Computer Society, 1992.

[14] K.D. Günther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communications*, 29(4), April 1981.

[15] Peter A.J. Hilbers and Johan J. Lukkien. Deadlock-free message routing in multicomputer networks. *Distributed Computing*, 3:178–186, 1989.

[16] C.R. Jesshope, P.R. Miller, and J.T. Yantchev. High performance communications in processor networks. In *Proc. 16th Intl. Symposium on Computer Architecture*, pages 150–157, 1989.

[17] P. Kermani and L. Kleinrock. Virtual Cut-Through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[18] S. Konstantinidou. Adaptive, minimal routing in hypercubes. In *Proc. 6th. MIT Conference on Advanced Research in VLSI*, pages 139–153, 1990.

[19] S. Konstantinidou and L. Snyder. The Chaos router: A practical application of randomization in network routing. In *Proc. 2nd. Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 21–30, 1990.

[20] Tom Leighton, Fillia Makedon, and Ioannis G. Tollis. A $2n - 2$ step algorithm for routing in an $n \times n$ array with constant size queues. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, 1989.

[21] D.H. Linder and J.C. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1):2–12, January 1991.

[22] P.M. Merlin and P.J. Schweitzer. Deadlock avoidance in store-and-forward networks. 1: Store-and-forward deadlock. *IEEE Transactions on Communications*, 28(3):345–354, March 1980.

[23] MP-1 family data-parallel computers, 1987. MasPar Computer Corporation, 749 North Mary Ave., Sunnyvale, CA.

[24] Michael Noakes and William J. Dally. System design of the J-Machine. In *Proc. Sixth MIT Conf. on Advanced Research in VLSI*, pages 179–194, 1990.

[25] Gustavo D. Pifarré, Luis Gravano, Sergio A. Felperin, and Jorge L.C. Sanz. Fully-adaptive minimal deadlock-free packet routing in hypercubes, meshes, and other networks. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 278–290, 1991.

[26] C. L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Seizovic, C. S. Steele, and W.-K. Su. The architecture and programming of the Ametek Series 2010 multicomputer. In *Proc. 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 33–36, 1988.

[27] Clark D. Thompson. Area-time complexity for VLSI. In *Proc. 11th Annual Symposium on Theory of Computing*, pages 81–88, 1979.

[28] Sam Toueg and Jeffrey D. Ullman. Deadlock-free packet switching networks. *SIAM Journal of Computing*, 10(3):594–611, August 1981.

[29] J. Yantchev and C.R. Jesshope. Adaptive, low latency, deadlock-free packet routing for networks of processors. *IEE Proc., Pt. E*, 136(3):178–186, May 1989.