A Set-to-Set Disjoint Paths Routing Algorithm in a Torus-Connected Cycles Network

Antoine BOSSARD
Graduate School of Science
Kanagawa University
2946 Tsuchiya, Hiratsuka
Kanagawa 259-1293, Japan

Keiichi KANEKO
Graduate School of Engineering
Tokyo University of Agriculture and Technology
2-24-16 Nakacho, Koganei
Tokyo 184-8588, Japan

Abstract

Torus-Connected Cycles (TCC) have been recently proposed as interconnection network for massively parallel systems. In order to increase system dependability and interprocessor communication performance, disjoint paths routing is critical. In this paper, we focus on the set-to-set disjoint paths routing problem which consists in select mutually node-disjoint paths between two sets of nodes. As a pre-requisite, we first propose a time-optimal solution to the 3-set-to-set disjoint paths routing problem in a $k$-ary $n$-dimensional torus, including its empirical evaluation. Consequently, we can solve the set-to-set disjoint paths routing problem in a $TCC(k,n)$ with paths of lengths at most $kn^2 + 2n$ in $O(kn^2)$ time.

keywords: torus, many-to-many, parallel system, fault-tolerance, dependable, supercomputer.

1 Introduction

Modern massively parallel systems includes thousands of CPU nodes. Processor connection has thus become a critical topic as it can easily provoke a bottleneck situation and severely harm global system performance. To address this issue, many advanced interconnection network topologies for such supercomputers have been introduced [5,9,10,11]. Yet, actual implementation of these topologies remains rare: theoretical advantages are overshadowed by complex hardware structuring and manufacturing. Facilitated implementation is precisely what targets the torus topology [13] (and with outstanding results as shown by the TOP500 list [14]), and furthermore the recently described torus-connected cycles (TCC) topology [1].

In this paper, we describe a solution to the set-to-set disjoint paths routing problem in a torus-connected cycles network. Precisely, given two sets of nodes $S,D$, this problem is about selecting mutually node-disjoint paths between the nodes of $S$ and those of $D$. This is an essential issue of parallel systems as it enables high-performance communication between processors. Effectively, by selecting mutually node-disjoint paths, notorious shared resources problems such as deadlocks, livelocks and starvations are guaranteed to never occur, hence enabling simultaneous routing over the network. In addition, disjoint paths routing significantly increases system dependability as one faulty node can break at most one path.

Related previous works include a hypercube set-to-set disjoint paths routing algorithm [6], a hypercube set-to-set cluster fault tolerant disjoint paths routing algorithm [8] and a star graph set-to-set fault tolerant disjoint paths routing [7]. In a TCC network, the node-to-node and node-to-set disjoint paths routing algorithms have been described in [2] and [3], respectively.

This paper has two main parts: first, the proposal of a 3-set-to-set disjoint paths routing algorithm in a torus, which is used in the second part addressing set-to-set disjoint paths routing in a TCC network.

2 Preliminaries

We recall in this section the first the torus topology definition as well as that of the TCC, and also give notations and definitions used hereinafter.

Definition 1. [4] An $n$-dimensional mesh has $n_i$ nodes on the $i$-th dimension ($k_i \geq 2, 0 \leq i \leq n$), inducing $k_0 \times k_1 \times \ldots \times k_{n-1}$ nodes in total. The address of a node $u$ has $n$ coordinates $(u_0, u_1, \ldots, u_{n-1})$. Two nodes $u, v$ are adjacent if and only if $u_i = v_i$ for all $i$ ($0 \leq i \leq n - 1$), except one, $j$, where either $u_j = v_j + 1$ or $u_j = v_j - 1$.

Definition 2. [4] A $k$-ary $n$-dimensional torus (a.k.a. $(k,n)$-torus) is an $n$-dimensional mesh with all $k_i$’s equal to $k$ and with wrap-around edges: two nodes $u, v$ are adjacent if and only if $u_i = v_i$ for all $i$ ($0 \leq i \leq n - 1$), except one, $j$, where either $u_j = v_j + 1 \pmod{k}$ or $u_j = v_j - 1 \pmod{k}$.

The degree of a $(k,n)$-torus is $n$ if $k = 2$, and $2n$ otherwise. Conveniently, we define the projection operator $[l]$ for torus nodes. In a $(k,n)$-torus, for a node $u = (u_0, u_1, \ldots, u_{n-1})$, define $u[\delta_0, \ldots, \delta_j] = (u_{\delta_0}, \ldots, u_{\delta_j})$ ($\delta_j < \delta_{j+1}, 0 \leq j \leq i-1, 0 \leq \delta_{n-j}$).
Figure 1: A 3-ary 2-dimensional torus (i.e. (3, 2)-torus).

Figure 2: A 3-ary 2-dimensional torus-connected cycles network TCC(3, 2).

\[ \delta_i, i \leq n - 1 \). For example, \( u = (3, 0, 5) \) and \( u[0 2] = (3, 5) \). Lastly, let “asc” (resp. “dsc”) denote the traversal of one torus dimension in ascending (resp. descending) order, that is incrementing (resp. decrementing) that dimension’s coordinate when routing. A (3, 2)-torus is illustrated in Figure 1.

**Definition 3.** A k-ary n-dimensional torus-connected cycles network TCC(k, n) has \( 2nk^n \) nodes. Each node a has a cluster ID \( c(a) = (a_0, a_1, \ldots, a_{n-1}) \) and a processor ID \( p(a) = p_a \), and the node consists of the pair \( (c(a), p(a)) \) where \( 0 \leq a_i \leq k - 1 \) and \( 0 \leq p_a \leq 2n - 1 \). Each node a has three adjacent nodes \( n_1(a), n_2(a) \) and \( n_3(a) \) defined as follows:

\[ n_1(a) = (c(a), (p_a + (-1)^{p_a}) \mod 2n) \]
\[ n_2(a) = (c(a), (p_a - (-1)^{p_a}) \mod 2n) \]
\[ n_3(a) = (a_0, a_1, \ldots, \{a[p_a/2] + (-1)^{p_a}\} \mod k, \ldots, a_{n-1}, p_a + (-1)^{p_a}) \]

A TCC(3, 2) is illustrated in Figure 2. If \( n = 1 \), the node degree is 2, and 3 otherwise. Each cycle (i.e. cluster) consists of \( 2n \) nodes. The cycle of a node a is denoted by \( C(a) \). A summary of the mesh, torus and TCC of several important topological properties is given in Table 1.

A path in a graph is an alternate sequence of nodes and edges \( a_1, (a_1, a_2), a_2, \ldots, a_k \). The length of a path \( p \) is its number of edges. We also use the notations \( a \rightarrow b \) and \( a \sim b \) to express an edge \( (a, b) \) and a path from a to b, respectively. In addition, \( a \Rightarrow b \) denotes in-cycle shortest-path routing from a to b.

### 3 3-set-to-set disjoint paths routing in a \((k, n)\)-torus

We describe in this section an algorithm to find three mutually node-disjoint paths connecting any three distinct source-destination pairs in a \((k, n)\)-torus. This is abbreviated as 3-S2S routing.

#### 3.1 Algorithm description

In a \((k, n)\)-torus, given a set \( S = \{s_1, s_2, s_3\} \) of three distinct source nodes and a set \( D = \{d_1, d_2, d_3\} \) of three distinct destination nodes \( (S \cap D \) not necessarily empty), we describe a routing algorithm finding three mutually node-disjoint paths \( s_i \sim d_j \) with \( s_i \in S \) and \( d_j \in D \).

The main idea of this algorithm is to reach from each source node a distinct 3-dimension position which will ensure paths disjointness. We distinguish four main steps as follows. Programming code in the functional language Scheme is given in Listing 1 with additional functions in Listing 2. The omitted function \texttt{torus-xor} simply marks the dimensions of different values for any two torus nodes.

**Step 1** Find three dimensions \( x, y, z \) such that the three \( d_i[xyz] \) (\( 1 \leq i \leq 3 \)) are distinct: \( d_1[xyz] \neq d_2[xyz] \), \( d_2[xyz] \neq d_3[xyz] \) and \( d_1[xyz] \neq d_3[xyz] \).

**Step 2** Generate all the six source-destination pairings: \( \{(s_1, d_1), (s_2, d_2), (s_3, d_3)\} \), \( \{(s_1, d_1), (s_2, d_3), (s_3, d_2)\} \), \( \{(s_1, d_2), (s_2, d_1), (s_3, d_3)\} \), \( \{(s_1, d_2), (s_2, d_3), (s_3, d_1)\} \), \( \{(s_1, d_3), (s_2, d_1), (s_3, d_2)\} \) and \( \{(s_1, d_3), (s_2, d_2), (s_3, d_1)\} \).

**Step 3** For each of all six pairings, generate the six orders possible: first connect \( s_1 \), then \( s_2 \), then \( s_3 \); first connect \( s_2 \), then \( s_1 \), then \( s_3 \); etc., in other words the permutations of the three source nodes.

**Step 4** For each source-destination pair of the current order of the current pairing, apply dimension-order routing (i.e. a dimension is fully processed after another) to all

<table>
<thead>
<tr>
<th>( n )-mesh</th>
<th>((k, n))-torus</th>
<th>TCC(k, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>( \prod_{i=0}^{k-1} k_i )</td>
<td>( k^n )</td>
</tr>
<tr>
<td>degree</td>
<td>2, 3, 4</td>
<td>2n</td>
</tr>
<tr>
<td>diameter</td>
<td>( \sum_{i=0}^{k-1} n_i )</td>
<td>( n[k/2] )</td>
</tr>
</tbody>
</table>

* Proven for \( k \) even.
the six possible dimension orders (i.e. permutations of the previously selected dimensions $x, y, z$ combined with the eight possible traversals for each dimension order ($\{asc, asc, asc\}, \{asc, dsc, asc\},$ etc.) until finding three mutually node-disjoint paths. Note that nodes included in both $S$ and $D$ are always paired together and thus connected by a path of length zero.

### 3.2 Correctness and complexities

We discuss in this section the correctness and worst case performance of the algorithm proposed in Section 3.1.

**Lemma 1.** We can always find three dimensions $x, y, z$ such that $d_1[xyz]$ (1 ≤ $i$ ≤ 3) are distinct: $d_1[xyz] \neq d_2[xyz]$, $d_2[xyz] \neq d_3[xyz]$ and $d_1[xyz] \neq d_3[xyz]$.

**Proof.** Take one dimension $x$ such that its value is different for $d_1$ and $d_2$: $d_1[x] \neq d_2[x]$. Because $d_1$ and $d_2$ are distinct, such dimension $x$ always exists. Then, we have three cases as follows: (1) if $d_3[x]$ is distinct with $d_1[x]$ and $d_2[x]$, then any other two distinct dimensions $y, z$ ($x \neq y, z$) will do and we have found three dimensions $x, y, z$; (2) if $d_3[x] = d_1[x]$, find a dimension $y$ such that $d_1[y] \neq d_3[y]$. Because $d_1$ and $d_3$ are distinct, such dimension $y$ always exists; (3) if $d_3[x] = d_2[x]$, find a dimension $y$ such that $d_2[y] \neq d_1[y]$. Because $d_2$ and $d_1$ are distinct, such dimension $y$ always exists. Hence, we have found two dimensions $x, y$ such that $d_1[xy], d_2[xy]$ and $d_3[xy]$ are distinct, so any other dimension $z$ ($z \neq x, y$) will do and we have found three dimensions $x, y, z$.

**Lemma 2.** The proposed algorithm is totally correct, $O(kn)$ optimal time and of maximum path length $kn$.

**Proof.** Once three dimensions $x, y, z$ have been found such that $d_1[xyz], d_2[xyz]$ and $d_3[xyz]$ are distinct, our algorithm simply tries to reach disjointly the distinct $x, y, z$ coordinates for each source-destination pair, which is possible by Menger’s theorem [12] (i.e. routing on only three dimensions $(x, y, z)$ on a $(3, 3)$-torus induces a node degree of $2n = 6$, and it is thus possible to find the three disjoint paths to $(x, y, z)$ from each source node). Since $d_1[xyz], d_2[xyz]$ and $d_3[xyz]$ are distinct, the remaining edges selected are guaranteed to be disjoint. The key for path disjointness is thus to reach the $x, y, z$ coordinates from the source node. We do it in a Cartesian manner by trying all possible dimension-order routings. Precisely (refer to Section 3.1 for additional details):

- 6 source-destination pairings possible;
- 6 orders per pairing possible;
- 3 source-destination pairs per pairing order;
- 6 dimension orders per source-destination pair possible;
- 8 traversals possible per dimension order;

So, in total, at most $6 * 6 * 3 * 6 * 8 = 5184$ trials are necessary to find a solution to the 3-S2S routing problem in a $(k, n)$-torus.

Because we apply dimension-order routing (one dimension after another, the order in which dimensions are processed vary though), the length of a longest path is $kn$ (maximum arity $k$ on each of the $n$ dimensions). Hence in total, the time complexity is $O(kn)$.

**Theorem 1.** In a $(k, n)$-torus, for two sets $S$ and $D$ with $|S| = |D| = 3$, we can find three mutually node-disjoint paths, each connecting a node of $S$ to a node of $D$, of lengths at most $kn$ in $O(kn)$ optimal time.

**Proof.** This can be directly deduced from Lemmas 1 and 2.

### 3.3 Average performance

We measure the average performance of the proposed 3-S2S routing algorithm in a $(k, n)$-torus through two experiments. For that, we have implemented the algorithm of Section 3.1, and we have inspected its average behaviour as detailed below.

For each value of $k$ the torus arity in $3 \leq k \leq 5$, and $n$ the torus dimension in $3 \leq n \leq 6$, we have used this implementation to solve 10,000 random instances of the 3-S2S routing problem in a $(k, n)$-torus. As a first experiment, given one pair $(k, n)$, for each of these problem instances, the length of a longest path is first recorded, then the maximum of all these recorded lengths is measured. See Figure 3. For reference, the theoretical maximum path length $kn$ is also represented.

As a second experiment, we have measured the average maximum path length, concretely recording the length of a longest path for each problem instance, then calculating the average of all these recorded lengths. See Figure 4. We can observe that in average, the path length measured stays below or converges towards the torus diameter $n \times k/2$ as $n$ increases, which is a positive indicator on the performance of our algorithm with respect to path length.

## 4 Set-to-set disjoint paths routing in a TCC

In this section, we propose a set-to-set disjoint paths routing algorithm in a TCC network. The algorithm is first described in Section 4.1 and complexities are then analysed in Section 4.2.

### 4.1 Algorithm description

In a $TCC(k, n)$, given a set $S = \{s_1, s_2, s_3\}$ of three distinct source nodes and a set $D = \{d_1, d_2, d_3\}$ of three distinct
Listing 1: Algorithm for 3-set-to-set disjoint paths routing in a $k$-ary $n$-dimensional torus.

(define (torus-3S2S k n S D)
  (let ([s1 (first S)] [s2 (second S)] [s3 (third S)] [d1 (first D)] [d2 (second D)] [d3 (third D)])
    ; Find three dimensions x, y, z such that $d_1[x]=d_2[y]$
    (let* ([x (- n (length (memq 1 (torus-xor d1 d2)))]
             [yz (let ([d1-x (list-ref d1 x)] [d2-x (list-ref d2 x)] [d3-x (list-ref d3 x)])
                   (cond [(= d3-x d2-x) ; pick y such that $d_2[y] 
eq d_3[y]$
                           (let* ([y (- n (length (memq 1 (list-change (torus-xor d2 d3) x 0))))]
                                 [z (if (= x 0) (if (= y 1) 2 1) (if (= y 0) (if (= x 1) 2 1) 0))])
                           '(,y . ,z))]
                 [(= d3-x d1-x) ; pick y such that $d_1[y] 
eq d_3[y]$
                           (let* ([y (- n (length (memq 1 (list-change (torus-xor d1 d3) x 0))))]
                                 [z (if (= x 0) (if (= y 1) 2 1) (if (= y 0) (if (= x 1) 2 1) 0))])
                           '(,y . ,z))]
                 [else ; $d_1[x], d_2[x], d_3[x]$ distinct, so any distinct dims will do for $yz$ (x excluded)
                           (if (= x 0) '(1 . 2)
                               (if (= x 1) '(0 . 2) '(0 . 1)))]
             )]
        yz)
    ; Generate the six sources-destination pairings possible
    (let ([pairings-raw '(( (,s1 . ,d1) (,s2 . ,d2) (,s3 . ,d3) )
                          ( (,s1 . ,d1) (,s2 . ,d3) (,s3 . ,d2) )
                          ( (,s1 . ,d2) (,s2 . ,d1) (,s3 . ,d3) )
                          ( (,s1 . ,d2) (,s2 . ,d3) (,s3 . ,d1) )
                          ( (,s1 . ,d3) (,s2 . ,d1) (,s3 . ,d2) )
                          ( (,s1 . ,d3) (,s2 . ,d2) (,s3 . ,d1) )
                          )]
        [ScapD (filter (lambda (s) (member s D)) S)] ; S \cap D
    [pairings (filter (lambda (p) (andmap (lambda (s) (member (cons s s) p)) ScapD))
                pairings-raw)] ; keep only pairings that pair nodes of $S$ also in $D$
    ; Generate all paths encodings for $xyz$ (other dims appended later)
    (encodings (apply append
                (map (lambda (dim-order)
                      (map (lambda (traversal)
                            '(( (,(first dim-order) . ,(first traversal))
                              (,(second dim-order) . ,(second traversal))
                              (,(third dim-order) . ,(third traversal)) ))
                      (asc asc asc) (asc asc dsc) (asc dsc asc) (asc dsc dsc)
                      (dsc dsc dsc) (dsc dsc asc) (dsc asc dsc) (dsc asc asc))
               (permutations '(,x ,y ,z)))))))
    ))
    )
    (let aux ([l pairings]) ; for each pairing
      (if (null? l) (error "No solution found.") ; never happens obviously
       (let* ([pairing (car l)]
                [pairing-all-orders (permutations pairing)]
                [pairing-orders pairing-all-orders])
        ; for each order of the current pairing
        ((let (pairing-order (car pairing-orders)))
         ; for each sd pair of the current pairing order, try all dimension-traversal combinations
         (let aux3 ([sd-pairs pairing-order] [res '()])
           (if (null? sd-pairs) res ; all three paths found, we are done
             ; else try to connect the current sd pair
             (let* ([sd (car sd-pairs)] [s (car sd)] [d (cdr sd)])
               (if (equal? s d) (aux2 (cdr sd-pairs) (cons (list s) res))
                 (let* ([other-dims ; encode all dims other than x, y, z
                         (for/list ([i n] #:unless (or (= i x) (= i y) (= i z))) 'i . opt)]
                           [path-dims ; s ~> d (encoded as dimension-traversal pairs)
                            (findf (lambda (dims) (are-disjoint (append (map (lambda (a) (list (car a) (cdr a))) (cdr sd-pairs))
                                             (cons (torus-route k s d dims) res)))
                                   (map (lambda (encoding) (append encoding other-dims)) encodings))]
                           [path-dims])
                 (if path-dims
                   ; try to connect the next sd pair of the current pairing
                   (aux2 (cdr sd-pairs) (cons (torus-route k s d path-dims) res))
                   ; else, try the next pairing order
                   (aux3 (cdr pairing-orders)))))
               ))
             )))))))
  ))
  ))
  ))
  )); Generate the six sources-destination pairings possible
  (pairings-raw ' ((,(s1 . ,d1) (,s2 . ,d2) (,s3 . ,d3) )
                    (,(s1 . ,d1) (,s2 . ,d3) (,s3 . ,d2) )
                    (,(s1 . ,d2) (,s2 . ,d1) (,s3 . ,d3) )
                    (,(s1 . ,d2) (,s2 . ,d3) (,s3 . ,d1) )
                    (,(s1 . ,d3) (,s2 . ,d1) (,s3 . ,d2) )
                    (,(s1 . ,d3) (,s2 . ,d2) (,s3 . ,d1) )
                    )]
    ))
    )); Generate all paths encodings for $xyz$ (other dims appended later)
    (encodings (apply append
                (map (lambda (dim-order)
                      (map (lambda (traversal)
                            '(( (,(first dim-order) . ,(first traversal))
                              (,(second dim-order) . ,(second traversal))
                              (,(third dim-order) . ,(third traversal)) ))
                      (asc asc asc) (asc asc dsc) (asc dsc asc) (asc dsc dsc)
                      (dsc dsc dsc) (dsc dsc asc) (dsc asc dsc) (dsc asc asc))
               (permutations '(,x ,y ,z)))))))
    ))

First, let us detail the following two special cases ($C$ represents a cycle of a TCC).

destination nodes, with the set $S \cap D$ not necessarily empty, we describe a routing algorithm that selects three mutually node-disjoint paths $s_i \sim d_j$ with $s_i \in S$ and $d_j \in D$.

The main idea of this algorithm is to spread out source (resp. destination) nodes into distinct cycles in order to apply the previously described 3-set-to-set disjoint paths routing algorithm in a $(k,n)$-torus. We shall distinguish several special cases and non-trivial patterns (i.e. source and destination nodes repartitions). Some trivial or repeating details shall be omitted for the sake of clarity.
Listing 2: Auxiliary functions: torus 1-dimension routing and path decoding.

```scheme
(define (torus-route k s d dims)
  ; decoding a path encoded as dimension-traversal pairs
  (let* ([dim (caar l)]
          [trav (cdar l)]
          [path (route-1-dim k (last res) d dim trav)])
    (aux v (cons v res)))))
```

Figure 3: Length of a longest path selected in a \((k, n)\)-torus experimentally.

**Special Case 1:** \(\exists C, S \cup D \subseteq C\)

Connect at least two source-destination pairs by in-cycle shortest-path routing in \(C\) for the closest source-destination node pairs. If all three pairs are connected, terminate. Otherwise, for the unconnected pair \(s_i, d_j\), select the path \(s_i \rightarrow n_3(s_i) \Rightarrow (c(n_3(s_i)), p_{d_j}) \rightarrow n_3((c(n_3(s_i)), p_{d_j})) = s' \Rightarrow (c(s'), p_{n_3(s(i)}) \rightarrow (c(n_3(d_j)), p_{s_i}) \Rightarrow n_3(d_j) \rightarrow d_j\). See Figure 5.

**Special Case 2:** \(\exists C, S \subseteq C, |D \cap C| = 2\)

Connect two source-destination node pairs inside \(C\) with in-cycle shortest-path routing. For the unconnected pair \(s_i, d_j\), select the path \(s_i \rightarrow n_3(s_i) \Rightarrow (c(n_3(s_i)), p_{d_j}) \rightarrow n_3((c(n_3(s_i)), p_{d_j})) = s' \Rightarrow (c(s'), p_{n_3(s(i)}) \rightarrow (c(n_3(d_j)), p_{s_i}) \Rightarrow n_3(d_j) \rightarrow d_j\). See Figure 6.

For all other cases, we shall apply the 3-S2S routing algorithm of Section 3.3 by spreading source nodes into distinct cycles, and likewise for destination nodes. The objective is to obtain three pairs of cycles, in other words three pairs of torus nodes, to apply the 3-S2S routing algorithm. So, for instance, in the case all source nodes are in distinct cycles, and likewise for destination nodes, we simply apply the torus 3-S2S routing algorithm to disjointly connect the three cycle pairs. Lastly, the obtained torus disjoint paths are converted back to TCC, still disjoint, paths with successive in-cycle routings; such conversion can be done with Algorithm 1.

Note that in-cycle routing in the last cycle may need to avoid other possible source or destination nodes, hence \(u \sim v\) instead of \(u \Rightarrow v\) (see for instance Figure 7). In practice, this is simply about choosing between clockwise and counter-clockwise for in-cycle routing, the algorithm being designed so that at least one of clockwise and counter-clockwise in-cycle routings is possible (i.e. not blocked by a source or destination node). An illustration is given in Figure 7. One should note that even if a source and destination node share the same cycle \(C\), no special processing needed as it is ensured that the trivial torus path of length zero \(C\) will be selected.

In some situations, it may be needed to do some preprocessing before applying the 3-S2S routing algorithm. We review these particular source-destination repartition patterns. Obviously, these patterns can be applied symmetrically to
source and destination nodes.

**Pattern 1:** \( \exists C, |S \cap C| = 2, |D \cap C| \leq 1 \)

Assume without loss of generality that \( S \cap C = \{s_1, s_2\} \). If \( n_3(s_1) \in C(s_3) \), select \( s_2 \rightarrow n_3(s_2) = s' \), otherwise select \( s_1 \rightarrow n_3(s_1) = s' \). Thus, we have three distinct cycles for all source nodes: \( C, C(s'), C(s_3) \). See Figure 8.

**Pattern 2:** \( \exists C, |S \cap C| = 2, |D \cap C| = 2 \)

Assume without loss of generality that \( s_3, d_1 \notin C \). Select a cycle \( C' \) with \( C' \notin \{C, C(s_3), C(d_1)\} \); its existence is obvious. We thus have three distinct cycles \( C, C(s_3), C' \) for source nodes, and three distinct cycles \( C, C(d_1), C' \) for destination nodes, and we can then apply 3-S2S torus routing.

**Pattern 3:** \( \exists C, S \subset C, |D \cap C| = 1 \)

Assume without loss of generality that \( d_1 \in C \) and that \( s_1 \) is the closest source node to \( d_1 \) on \( C \). If \( \exists C', D \cap C' = \{d_1, d_2\} \), select \( d_1 \rightarrow n_3(d_1) = d' \) if \( n_3(d_1) \notin C \) and \( d_2 \rightarrow n_3(d_2) = d' \) otherwise. We thus have three distinct cycles \( C, C(d_1), C(d') \) for destination nodes. If no such cycle \( C' \) exists, we directly have three distinct cycles \( C, C(d_1), C(d_2) \) for destination nodes. Select \( s_2 \rightarrow n_3(s_2) = s'_2 \) and \( s_3 \rightarrow n_3(s_3) = s'_3 \). We thus have three distinct cycles \( C, C(s'_2), C(s'_3) \) for sources nodes, and we can then apply 3-S2S torus routing. Remember that we are ensure that the trivial path \( C \) of length zero will be selected.

As shown in Figure 9, when converting paths, say for instance in a cycle with two destination nodes like \( C' \) here, we must pay attention not to include any other destination node when performing in-cycle routing: \( p : s'_3 \rightarrow d_2 \) selected such that \( d_1 \notin p \). Here, \( p \neq (s'_3 \rightarrow d_2) \).

**Pattern 4:** \( \exists C, S \subset C, D \cap C = \emptyset \)

Select the three edges \( s_1 \rightarrow n_3(s_1) = s'_1, s_2 \rightarrow n_3(s_2) = s'_2 \) and \( s_3 \rightarrow n_3(s_3) = s'_3 \). The 3-S2S torus routing is applied and thus three disjoint paths \( p_1, p_2, p_3 \) respectively starting from \( C(s'_1), C(s'_2) \) and \( C(s'_3) \) are obtained. From now on, refer to Figure 10 for more clarity. If \( C \) not included in one of these three torus paths, they can be directly converted back to TCC paths.

We can now assume without loss of generality that \( C \) is included in one of \( p_1, p_2, p_3 \). If \( C \) in \( p_1 \) or \( p_3 \), in-cycle routing in \( C \) is possible, but not if \( C \) in \( p_2 \). So, we assume \( C \) included in the path \( p_2 \), thus of the form \( p_2 : C(s'_2) \rightarrow C \rightarrow C'' \rightarrow \ldots \), with \( C'' \) another distinct cycle, adjacent to \( C \) and obviously distinct from \( C(s'_1) \) and \( C(s'_3) \). Depending on the dimension differing between \( C(s'_2) \) and \( C \), and that between \( C \) and \( C'' \), additional post-processing (rerouting) may be needed. Let \( w \in C \) be the unique node satisfying \( n_3(w) \in C'' \). If there is an in-cycle path \( s_2 \rightarrow w \) that includes neither \( s_1 \) nor \( s_3 \), then no rerouting is necessary, and paths can be converted back to TCC paths directly. Now, if there is no such \( s_2 \rightarrow w \) path, rerouting is needed. We detail below the three rerouting situations.

**Case \( C'' \) included on \( p_2 : C(s'_2) \rightarrow C \rightarrow C'' \rightarrow \ldots \)**

The path \( p_2 \) is modified as follows. Discard the sub-path \( C(s'_2) \rightarrow C \rightarrow C'' \) of \( p_2 \). Select the edge \( C(s'_2) \rightarrow C' \).

**Case \( C'' \) included on \( p_1 : C(s'_1) \rightarrow \ldots \)**

First, \( p_1 \) is modified as follows. Discard the sub-path \( C(s'_1) \rightarrow C' \) of \( p_1 \). Select the edge \( C(s'_2) \rightarrow C' \). Then,
Case $C' \notin p_1 \cup p_2$

Assume $C'$ included on $p_3$. First, $p_3$ is modified as follows. Discard the sub-path $C(s'_3) \sim C'$ of $p_3$. Select the edge $C(s'_3) \rightarrow C$. Then, $p_2$ is modified as follows. Discard the edge $C(s'_2) \rightarrow C$ of $p_2$, and the edge $s_3 \rightarrow s'_3$.

Assume $C'$ not included on $p_3$. First, $p_3$ is modified as follows. Select the path $C(s'_3) \rightarrow C' \rightarrow C(s'_2)$. Then, $p_2$ is modified as follows. Discard the edge $C(s'_2) \rightarrow C$ of $p_2$, and the edge $s_3 \rightarrow s'_3$.

In other words, the torus path for $C(s'_2)$ is rerouted to the torus path initially for $C(s'_3)$, and the torus path for $C(s'_2)$ is rerouted to the torus path initially for $C(s'_1)$.

We conclude Pattern 4 by showing that the case $D \subset C'$ is solved similarly. Select the three edges $d_1 \rightarrow n_3(d_1) = d'_1$, $d_2 \rightarrow n_3(d_2) = d'_2$ and $d_3 \rightarrow n_3(d_3) = d'_3$. The 3-S2S torus routing is applied and thus three disjoint paths $q_1, q_2, q_3$ respectively ending with $C(d'_1), C(d'_2)$ and $C(d'_3)$ are obtained. Again, if $C'$ not included in one of these three torus paths, or if $C'$ included in either $q_1$ or $q_3$, no additional post-processing is required.

So, we assume $C' \in q_2$, thus $q_2$ of the form $q_2 : C(d'_3) \rightarrow C' \rightarrow C''' \rightarrow \ldots$, with $C'''$ another distinct cycle, adjacent to $C'$ and obviously distinct from $C(d'_1)$ and $C(d'_3)$. Note that we still assume $C$ included in the path $p_1$ to discuss the trickiest situation, thus excluding the easier case $\exists d'_1 \in C(s'_2)$. If $\exists d'_1 \in C(s'_3)$, refer to Figure 11 otherwise refer to Figure 12. We solve the problem as in Pattern 4 with $C'''$ getting the role of $C'$ and $C''''$ that of $C''$. The case $D \subset C'$ is identical to this case $D \subset C'$, with the sub-case $C''''$ included on $C(d'_2) \rightarrow C' \rightarrow \ldots$ never occurring.

Figure 9: Illustrating one possible Pattern 3 situation. $C'$ in-cycle routing $s'_3 \sim d_2$ is not shortest.

Figure 10: Illustrating the basic situation of Pattern 4.

Figure 11: Illustrating Pattern 4 with $D \subset C'$ and $d'_1 \in C(s'_3)$.

Figure 12: Illustrating Pattern 4 with $D \subset C'$ and $d'_i \notin C(s'_3)$ ($1 \leq i \leq 3$).

4.2 Complexities

We discuss here the worst-case time complexity of the algorithm of Section 4.1 as well as the maximum length of a selected path.

Theorem 2. In a TCC($k, n$), given two sets $S, D$ of three distinct nodes each, we can find three mutually node-disjoint paths linking the nodes of $S$ to those of $D$, of lengths at most $kn^2 + 2n$ in $O(kn^2)$ time.

Proof. The correctness of the algorithm proposed in Section 4.1 has been proved for each case and pattern. Regarding the maximum length of a selected path, it can be estimated as follows. First, by Theorem 1 the maximum length of the torus paths is $kn$. Then, in-cycle routing is conducted inside each cycle corresponding to a node of these torus paths. Except for the last cycle of a torus path, each in-cycle routing
requires at most \( n \) edges; we recall that each cycle includes \( 2n \) edges and that shortest-path routing in-cycle is applied. Inside the last cycle of a torus path, shortest-path routing is not guaranteed, thus inducing at most \( 2n - 2 \) edges (i.e. assuming one source or destination node blocks the shortest path). Lastly, at most 1 edge is required to connect a source node to a distinct cycle, the same holding for a destination node. Therefore, in total, the maximum path length is \( kn \times n + (2n - 2) + 2 = kn^2 + 2n \).

By Theorem 1, torus 3-S2S routing takes \( O(kn) \) time. The conversion of one torus path (\( O(kn) \) cycles) back to the corresponding TCC path as per Algorithm 1 takes \( O(kn \times n) \) time, which is thus the dominant time complexity of this algorithm.

\[ \square \]

5 Conclusions

Disjoint paths routing is critical for system performance and dependability. In this paper, we have first proposed a time-optimal algorithm solving the 3-set-to-set disjoint paths routing problem in a \( k \)-ary \( n \)-dimensional torus. We have formally established the worst-case time complexity and maximum path length of this algorithm. Then, we have conducted experiments to inspect the practical behaviour of this torus algorithm, and observed that in average, the lengths of the selected paths either converge towards the torus diameter, or stay below it, which is a positive performance indicator. Building upon this result, we have next described a set-to-set disjoint paths routing algorithm in a TCC(\( k, n \)) that selects paths of lengths at most \( kn^2 + 2n \) in \( O(kn^2) \) time, thus significantly improving on existing solutions such the one provided by a maximum flow algorithm. Still aiming at improving system fault-tolerance, addressing cluster faults when routing is a very meaningful future work due to the cluster-based nature of the TCC network topology.

Acknowledgements

The authors sincerely thank the reviewers for their comments. This study was partly supported by the TEPCO Memorial Foundation.

References


