An Optimal Inherently Stabilizing Algorithm for Routing over All Node-Disjoint Paths in Exchanged Hypercubes

Thamer Alsulaiman and Mehmet Hakan Karaata

***Abstract*—Two paths between a source node and a destination node in a network are node- disjoint if they do not share any nodes except the end points. Node-disjoint paths have numerous uses in distributed systems including ways to deal with lost, damaged or altered messages during delivery. Many topologies such as hypercube, star networks, and their variants have been proposed, providing multiple disjoint paths between a pair of endpoints. The exchanged hypercube is a new topology that is obtained by systematically removing edges from a binary hypercube. Exchanged hypercube topology increases scalability and relative cost of the networks by reducing the number of edges per node. In this paper, we propose a distributed algorithm that is both stabilizing and inherently stabilizing to route messages over all node-disjoint paths in an exchanged hypercube network.**

***Index Terms*—Distributed algorithms, exchanged hypercube, node-disjoint paths, stabilization.**

I. Introduction

In distributed systems, communication delays and throughput of the interconnection network are important factors on the overall performance of the system [1]. In order to minimize communication delays and to increase network throughput, system components can be connected via a network providing node-disjoint paths. Two paths from a source to a destination are said to be node-disjoint if they share no common nodes except for the source and the destination. The (all) node-disjoint paths problem is a fundamental problem with many applications in diverse areas including VLSI layout [1], reliable network routing [2], [3], secure message transmission [4], and network survivability [5]. For instance, node-disjoint paths can be used for perfectly secure transmission as follows. The simple expedient of breaking up data into several shares and sending them along the disjoint paths makes it difficult for an adversary with bounded eavesdropping capability to intercept a transmission or tamper with it. Alternatively, the same crucial message can be sent over multiple node-disjoint paths in a network that is prone to message losses to avoid omission failures, or information on the re-routing of traffic along non-faulty disjoint paths can be provided in the presence of faults in some disjoint paths. Other applications of disjoint paths include network coding to provide 1+N protection against single link failures in optical hypercube networks [5], where N is the dimension of the network, and VLSI layout [6].

Manuscript received July 16, 2014; revised December 12, 2014. This work supported by Kuwait University Research Grant EO 01/11.

Thamer Alsulaiman is with the University of Iowa, 14 MacLean Hall, Iowa City, IA 52242-1419, US (email: thamer.mohsen@gmail.com)

Mehmet HakanKaraata is with Kuwait University, Safat 13090, Khaldiya, Kuwait (e-mail: karaata@eng.kuniv.edu.kw).

Due to the continuous increase in the number of nodes included in massively parallel systems, the probability of faults is constantly increasing. For this reason, it is critical to find mutually disjoint paths in order to establish communication routes under such a faulty environment as proposed in [7]. The presence of node-disjoint paths in a network can be used to reduce delays and increase system throughput; however, it reduces the scalability of the network due to increased connectivity.

A desirable network topology should provide a reasonable balance between the number of links and the number of nodes in the network, while providing other desirable properties such as ease of routing, network embeddings, and fault tolerance [8]. In addition, as the number of links per node is restricted due to hardware limitations, the underlying topology used needs to minimize the number of links per node while retaining a small diameter to remain efficient. Several network topologies have been proposed with some of these desirable properties. For instance, Hypercubes and star graphs are rich, recursively structured and symmetrical interconnection topologies for multiprocessor systems with many desirable fault tolerance characteristics. However, in star graphs, the number of nodes needs to be the factorial of an integer. In practical terms, this is a severe restriction on the sizes of systems that can be built; there is a large gap between the numbers $(n-1)!$ and $n!$. The n-dimensional hypercube (n-cube) with 2n nodes and $n2n-1$ links proposed by Saad and Schulz [9] is a topology with many properties such as the presence of n disjoint paths between each pair of distinct nodes. Extensive research has been undertaken on the n-cube, such as routing, fault tolerance, and embeddings [9]-[11]. C. N. Lai presents an algorithm for constructing n disjoint paths of optimal total length between a source and a destination nodes in an n-cube, where n is the dimension of the hypercube [7]. However, the n-cube scales too rapidly as dimension n grows, i.e., the number of links is high relative to the number of nodes. In order to overcome the scalability problem of the simple interconnection networks such as the hypercube, star and ring topologies have been superseded by more complex variations of the n-cube such as Gaussian Hypercube, perfect hierarchical hypercube and Reduced Hypercube [7], [12], [13]. These variants of the n-cube are produced by removing some of the links of a regular n-cube. These interconnection networks can connect many nodes while keeping a small diameter and low degree compared to hypercubes of the same size. The removal process affects some of the topological properties such as ease of routing in the presence of faults.

The Exchanged Hypercube is a topology obtained by systematically removing links from an n-cube to reduce interconnection complexity while maintaining several essential properties of the n-cube, such as being Hamiltonian,

DOI: 10.7763/JACN.2015.V3.138

optimally embedding linear arrays and rings, and embedding meshes and trees with reasonable efficiencies. A spanning tree of the Exchanged Hypercube referred to as the Extended Binomial Tree provides the necessary framework for solving many applications such as broadcasting, prefix sum computing and load balancing in the Exchanged Hypercube. In addition, the Exchanged Hypercube has a number of desirable properties such as small diameter, low degree, fault tolerance, strong connectivity, recursive construction, partition capability and low latency. These properties enable it to serve as a cost effective interconnection topology for constructing fault tolerant networks in a peer-to-peer (P2P) environment. A system is referred to as an inherently stabilizing system iff neither arbitrary initialization nor transient faults affecting the configuration of the system processes have any effect on the execution of the algorithm or its progress. However, such a system offers no guarantee if the transient faults affect the communication links or message buffers. While an inherently stabilizing system continues its correct execution without any delay in the event of a transient fault or after starting in an arbitrary initial configuration, a stabilizing system ensures correct execution of the algorithm only upon stability is reached after a delay. On the other hand, an inherently stabilizing system may neither mask nor tolerate transient faults affecting communication links and message buffers of the system, whereas a stabilizing system can cope with these faults in addition to those affecting the configuration of the processes. As a result, an inherently stabilizing system may not be stabilizing and vice versa. The first inherently stabilizing algorithm is proposed in for routing in hypercube networks.

The paradigm of inherent stabilization resembles that of snap stabilization. A system is snap stabilizing if after the system starts, it behaves as per its specification without any delay, regardless of the system configuration, in the absence of faults. However, such systems provide no guarantees in the event of transient fault (s) after the system starts. On the other hand, an inherently stabilizing system always behaves correctly without any delay in the presence of faults even after transient faults take place. The inherent stabilization property is a stronger property than the snap-stabilization property.

The problem of finding disjoint paths in various topologies including OTIS networks, incomplete WK-recursive networks, and a level block of generalized hierarchical completely connected networks are available. However, a stabilizing or inherently stabilizing algorithm for finding disjoint paths in Exchanged Hypercubes is not available in the literature.

In this paper, we propose a stabilizing and inherently stabilizing algorithm for routing over all node-disjoint paths between any two nodes in an exchanged hypercube. In particular, the proposed routing algorithm allows source process S to send k messages to destination process D, where S $\neq$ D in at most d(S,D)+4 rounds in the absence of transient faults in an exchanged hypercube such that each message traverses a distinct node-disjoint path in reaching the destination process, where k is the number of available paths in the graph, and d(S,D) denotes the (shortest) distance in hops between node S and D. A round refers to the minimal execution in the system in which each process executes all its enabled actions at the beginning of the round, and all the messages sent by these executed actions are delivered to the neighboring destination message buffers. The proposed algorithm is stabilizing and inherently stabilizing, due to being inherently stabilizing, transient faults affecting the system configuration, excluding system buffers, are masked and have no effect on the correct execution of the algorithm. Also, since it is stabilizing, the algorithm eventually recovers from transient faults affecting system buffers. Note that the proposed algorithm can be viewed as an extension of the protocol presented by Sinanoglu *et al.* [14] for the hypercube topology to the Exchanged Hypercube topology.

II. PRELIMINARIES

Let Exchanged Hypercube *EH* (*s*, *t*)=(*V*, *E*) be an undirected graph, where $s \geq 1$ and $t \geq 1$, *V* is the vertex set, *E* is the edge set, and *s* and *t* are dimensions of subcubes in the Exchanged Hypercube. The *id* of node *v* in *EH* (*s*, *t*) is a bit sequence $a_{s-1}\ldots a_0 b_{t-1}\ldots b_0 c$ where a_i, b_j, $c \in \{0, 1\}$ for $i \in [0, s\text{-}1]$, $j \in [0, t\text{-}1]$. $v[i]$ denotes the i^{th} bit in the id of node v.v $[x{:}y]$ denotes a subsequence of $idv \in EH$ (*s*, *t*) between two bit positions *x* and *y*, where $s+t \geq x \geq y \geq 0$.Observe that each $idv = a_{s-1}\ldots a_0 b_{t-1}\ldots b_0 c$ is composed of three subsequences, namely, $a_{s-1}a_{s-2}\ldots a_0$, $b_{t-1}b_{t-2}\ldots b_0$, and *c* referred to as *s-subsequence*, *t-subsequence* and *dummy bit*, respectively. The least significant bit *c* (*v* [0]) of each node *v* is referred to as a *dummy bit*. The dummy bit value of a node determines whether the node is of *s-type* or *t-type*. In particular, if *c*=0 holds for node *v* in *EH*(*s*, *t*), node *v* is called *s-type*; otherwise, *t-type*.

An Exchanged Hypercube *EH* (*s*, *t*) consists of 2^t *s-subcubes* each of which containing 2^s *s-type* nodes with the same *t-subsequence*, and 2^s *t-subcubes* each of which containing 2^t *t-type* nodes with the same *s-subsequence*. In an Exchanged Hypercube, the number of *s-type* and *t-type* nodes are the same.Each *s-type* node is connected to a *t-type* node by an edge referred to as a *dummy edge* such that their bit sequences are the same except for their dummy bits. An edge (v_1, v_2) is included in *E* if v_1 and v_2 are of *t-type*, their *hamming distance* is one and their *s-subsequences* are the same. The *hamming distance* between two bit sequences refers to the number of positions at which the corresponding bits are different. An Exchanged Hypercube of dimensions (1, 2) is shown in Fig. 1. The figure illustrates the concepts given above where dummy edges are shown by dashed lines whereas edges between same type nodes are shown by solid lines.

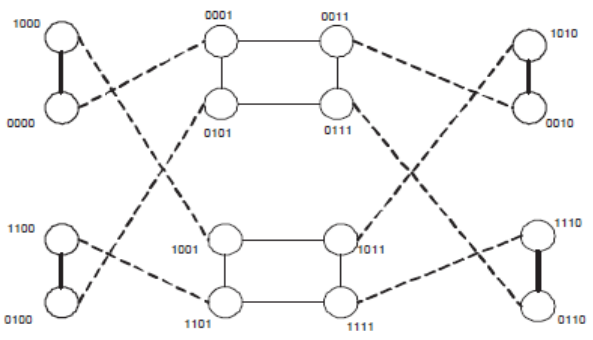

Fig. 1. An exchanged hypercube of dimensions (1, 2).

III. Input, Output and Actions

We assume that there is a separate protocol called the *application protocol*, which uses the proposed node-disjoint paths algorithm to send a sequence of messages from a source process to a destination process over all *node-disjoint* paths. The *node-disjoint* paths algorithm maintains two implicit buffers for each process referred to as the implicit *input* buffer and the implicit *output* buffer. These two buffers are also referred to as *interface* buffers, and are used to implement the interface between the application protocol and the *node-disjoint* paths algorithm.

The *node-disjoint* paths algorithm maintains two implicit buffers for each process referred to as the implicit *input* buffer and the implicit *output* buffer. These two buffers are also referred to as *interface* buffers, and are used to implement the interface between the application protocol and the *node-disjoint* paths algorithm.

When the application protocol is to send a set of messages from source process *S* to destination process *D*, it places a sequence of *k* messages $M=m_0, m_1, m_{2...}m_{k-1}$destination process id *D*, and the exchanged hypercube dimensions *s* and *t* in the input buffer of process *S*. This input is removed from the input buffer of process *S*, by the *node-disjoint* paths algorithm and the *k* messages are routed by the proposed protocol over *k* disjoint paths from *S* to *D*. Upon arrival of each message *m* at the destination process *D*, the message is placed in the *output* buffer of destination process *D*so that the message is ready to be collected by the application protocol.

The *input* and *output* buffers are maintained by each process of the proposed system to allow each process to be the source process or the destination process. The *input* buffer of each process contains at most a single sequence of *k* messages at any point in time. Analogously, the *output* buffer of each process contains at most one message. Furthermore, in every *x*+4 rounds, the application protocol allows each process to initiate a single message sequence consisting of at most *k* messages, where *x* is the diameter of the exchanged hypercube *EH*(*s*, *t*).

In addition to the interface buffers, each system process maintains an implicit message buffer which holds at most *k* incoming messages until these messages are received by the process.

IV. Basis of the Algorithm

Source *S* and destination *D* (or any other process) in an *s*-subcube and in a *t*-subcube has *s*+1 and *t*+1 neighbors, respectively. Since *s* and *t* are not necessarily equal, *S* and *D* may not have the same number of neighbors. Therefore, the number of available disjoint paths *k* between *S* and *D* depends on the smaller of the number of neighbors of *S* and *D*. The maximum number of disjoint paths *k* between nodes *S* and *D*, varies depending on the types of nodes *S* and *D* as follows:

$$k=\begin{Bmatrix} s+1 & \text{if } S \text{ and } D \text{ are of } s\text{-type} \\ t+1 & \text{if } S \text{ and } D \text{ are of } t\text{-type} \\ \min\{s,t\}+1 & \text{if } S \text{ and } D \text{ are not of the same type} \end{Bmatrix}$$

When the input buffer of source process *S* contains an input *M* from the application protocol, the routing protocol removes the input from the buffer by executing the corresponding guard, where $M=<m_0, m_1, ..., m_{k-1}>$ is a sequence of *k* messages, and *D* is the destination id. Then, source node *S* maps each of its neighbors to a distinct neighbor of the destination, using function *map* (). After the mapping is completed, the routing protocol at the source sends each message m_i, for $0\le i<k$, to a distinct neighbor of the source with appropriate parameters. Then, the routing protocol routes each message m_i, $0\le i<k$, between a neighbor *v* of the source and a neighbor *w* of the destination that are mapped over a path disjoint from the paths traversed by other messages. Finally, when a message reaches a neighbor of the destination, it is forwarded to the destination to complete the routing.

Each process in an *s*-subcube has *s* neighbors in the same *s-subcube*, and a single neighbor in a *t-subcube* called a *dummy neighbor*.

The routing over all disjoint paths between *S* and *D* depends on whether *S* and *D* are of the same or different type (*s-type* or *t-type*).

We now describe the routing process when *S* is of *s-type* and *D* is of *t-type* and $s\le t$ holds. We know that there are *snon-dummy* paths and a single *dummy path* between *S* and *D*. A path that contains the dummy neighbor of the source as the second process towards *D* is referred to as a *dummy path*, and *non-dummypath* otherwise. We describe the routing of messages from *S* to *D* over *s*+1 paths in seven phases. First, the routing begins when *S* sends each of the *s* messages to a distinct neighbor *v* in the same *s-cube* by flipping a distinct position in its *s-subsequence* and the dummy bit. Below we describe the routing of each one of these messages to destination *D* through the mapped neighbors *v* and *w* of *S* and *D*, respectively. Second, upon arrival of a message at a *non-dummy* node *v*, *non-dummy* neighbor *v* of *S* flips its dummy bit to route the message to a node in a *t-subcube*. Observe that since the neighbors of *S* in the same subcube have distinct *s-subsequences*, after flipping its dummy bit, each message reaches a distinct *t-subcube*. On the other hand, upon reciept of a message by the dummy neighbor of *S*, the message is already in a *t-cube*. Third, each message is routed in the reached *t-subcube* until the *t-subsequence* of the process reached by the message is equal to that of *w*. Fourth, by flipping the dummy bit, each message reaches a distinct *s-subcube*. Observe that, when routing is complete in the *t-cubes*, *t-subsequences* of the reached nodes are the same as those of the corresponding neighbors of the destination and therefore are unique. As a result, after flipping the dummy bit, each message reaches a distinct *s-subcube* (addressed by the unique *t-subsequence*). Fifth, each message is routed in the reached *s-subcube* until the *s-subsequence* is equal to that of the neighbor *w* of the destination. Sixth, the dummy bit is flipped to reach *w* except for the message that is mapped to the dummy neighbor of *D*. The message mapped to the dummy neighbour of the destination reaches destination *D* after the dummy bit-flip in the sixth step while others reach neighbors of destination *D*. Finally, if the message did not reach the destination, it is sent to the destination. Observe that the routing when *S* is of *t-type* and D is of *s-type* where $t\le s$ is analogous. We illustrate the above concepts using Fig. 2 showing *k* disjoint paths $P_0, P_1, P_2, ...P_{k-1}$ from source *S* to destination *D*. In the figure, $v_0, v_1, ..., v_{k-1}$.

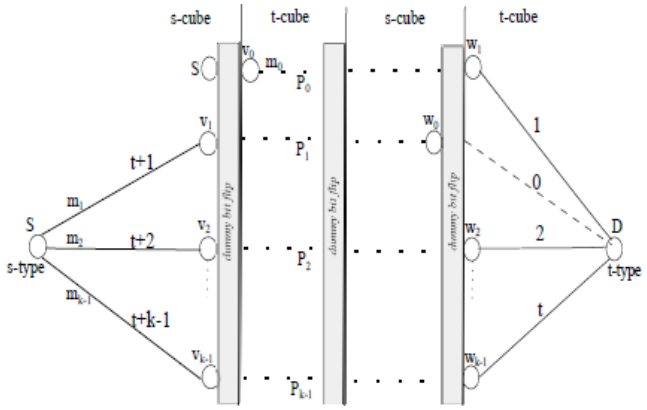

Fig. 2. Routing in an exchanged hypercube where source *S* is *s*-type and destination *D* is *t*-type.

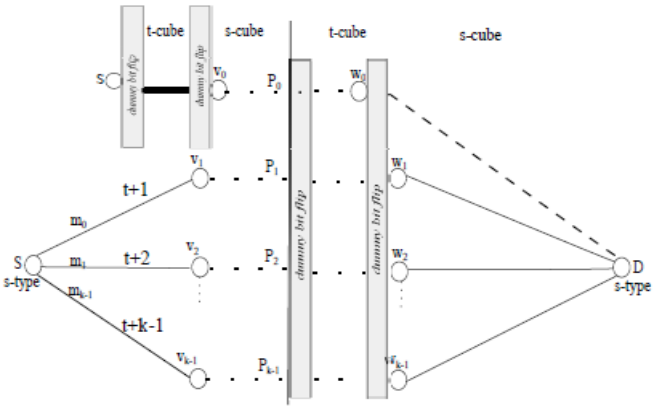

Fig. 3. Routing in an exchanged hypercube where source *S* and destination *D* are both *s*-type.

We now describe the routing when *S* and *D* are of both *s-type*. We know that there are *snon-dummy* paths and a single *dummy path* between *S* and *D*. We first describe the routing of a message over a *non-dummy* path from *S* to *D* through the mapped neighbors *v* and *w* of *S* and *D*, respectively, in six phases. First, routing begins when *S* sends each message to a distinct neighbor *v* in the same subcube by flipping a distinct position I n its *s-subsequence*. Second, the message is routed from node *v* to node *v*' where *v*' is in the same *s-subcube* as *S* such that the *s-subsequence* of *v*' is the same as that of *w*. Third, the dummy bit of *v*' is flipped to enter a *t-subcube* where each message reaches a distinct *t-cube*. Fourth, the bits in the *t-subsequence* that are different from the *t-subsequence* of *w* are flipped to carry out the routing in the reached *t-subcube*. Fifth, the dummy bit is flipped to reach *w* in the *s-subcube* containing *D*. Sixth, the last remaining bit in *w* that is different from *D* is flipped to reach *D*.

We now describe the routing of a message over a dummy path between source *S* and destination *D* through the mapped dummy neighbors of *v* and *w* of *S* and *D*, respectively, in six phases. First, starting from *S*, the dummy bit is flipped to reach *v* in a *t-subcube*. (Then, the routing needs to continue in an *s-cube* different from the one containing *S* and the initial segments of other *non-dummy* paths. To implement this, the dummy path is extended in the following manner after reaching a *t-subcube*). Second, a *sorted* bit of *v* in its *t-subsequence* is flipped to move to another process in the same *t-subcube*. A bit position *i* is said to be *sorted* if $v[i]=w[i]$, and *unsorted* otherwise, where *v* is the current process id and *w* is the id of the mapped destination neighbor. If no sorted bit exists, an unsorted bit is flipped. Third, the dummy bit is flipped to reach an *s-subcube* different from the one containing *S*. Since these two steps lead the message to a subcube of the same type as *S* but different than the one containing *S*, the second and third steps of the routing are referred to as *subcubeswitch*. In the next two phases, routing in *s-cubes* followed by routing in *t-cubes* are performed as in the case of routing over a *non-dummy* path between *S* and *D* of*s-type* as described above to reach w_0 which is a dummy neighbor of *D*. Finally, the dummy bit is flipped to reach *D*. It is easy to see the case where *S* and *D* are of *t-type* is analogous. We use Fig. 3 to illustrate the concepts mentioned in the above paragraph. Fig. 3 is similar to Fig. 2 but it has a dummy path P_0 for which routing is slightly different from that of *non-dummy* paths.

V. Mapping

Map (*S*, *D*, *i*) is any function that satisfies all of the following conditions,

1) $\sum d(i, j)$ is minimal, $i \in N_s \wedge j$=map (*S*, *D*, *i*), where N_s denotes the set of neighbors of the source *S*.
2) If $s < t$, each neighbor of the source is mapped to a distinct neighbor of the destination. On the other hand, if $t \leq s$ then each neighbor of the destination is mapped to a distinct neighbor of the source.
3) If *j*=map(*S*, *D*, *i*) for $i \in Ns$, then $j \in N_D$, where N_D denotes the set of neighbors of the destination *D*.

Condition 1) guarantees that the mapped set of nodes do not intersect in the resulting routing scheme since the generated routes follow the shortest path overall. It also maintains the disjointness of the algorithm.

Condition 2) and 3) facilitates routing the messages without generated paths intersecting one another.

VI. Routing

Source *S* sends each of the *k* messages of message sequence *M* to a distinct neighbor *i* of *S* with appropriate parameters. Each message contains the following parameters: message m_i, source id *S*, destination id *D*, *N* (*D*, *map*(*S*, *D*, *i*)), *fPhase*(*S*, *D*, *i*) and *map* (*S*, *D*, *i*). *N*(*D*, *map*(*S*, *D*, *i*)) denotes the neighbor of destination *D* that i^{th} neighbor of *S* is mapped to and *map*(*S*, *D*, *i*) denotes the bit-flip position in destination id *D* flipped to obtain the id of the neighbor of *D* that i^{th} neighbor of *S* is mapped to. Function *fPhase*() is described below. After determining all the parameters, source process *S* sends each message m_i, $0 \leq i < k$, to its i^{th} neighbor determined using *N*(*S*,*i*) by executing *send*().

The routing between the source and the destination takes place in two consecutive phases. The routing phase is said to be *s-phase* and *t-phase* when routing takes place in *s-cubes* and *t-cubes*, respectively. The routing sequence is either *s-phase* routing followed by *t-phase* routing or *t-phase* routing followed by *s-phase* routing. The selection of the proper sequence of phases depends on the destination type: if the destination is of *s-type*, the routing phase sequence is *s-phase* then *t-phase*. If the destination is of *t-type*, the routing phase sequence is *t-phase* then *s-phase*.

We need the following definitions to facilitate the description of the algorithm. *fPhase*(*S*, *D*, *i*) returns 0 to

indicate that a subcube switch is required prior to the first *phase*, returns 1, otherwise. This subcube switch is required only for the message routed through the dummy neighbor of the source when the source and the destination are of the same type.

$$fPhase(S,D,i)=\begin{Bmatrix}0 \text{ if } S[0]=D[0]\wedge i=0\\ 1 \text{ Otherwise}\end{Bmatrix}$$

We now describe the routing carried out between a neighbor of *S* and a neighbor of *D* that are mapped. Upon receipt of a message, if process *p* is the destination, the message will be output to the application protocol. If process *Pd* is not a neighbor of destination *D*, then an error in *Pd*, *f*, or *D* is encountered and the routing is terminated. If process *p* is a neighbor of the destination, the message is sent to the destination. Otherwise, process *p* is an intermediate process, and process *p* determines the next process to which the message is to be forwarded, and then forwards it to the decided neighboring process.

The next process to forward the message is determined by first deciding the current phase of the routing. We use function *routPhase*() as defined below to determine the order of phases, *s-phase* and *t-phase*, in the routing.

$$resultPhase(S,D,curPhase)=\begin{cases} CT(S) & if\ (S[0]=D[0]\wedge \text{curPhase}=1)\ or\\ & (S[0]\neq D[0]\wedge \text{curPhase}=2)\\ \neq CT(S) & otherwise\end{cases}$$

where, function *CT*(*S*) returns the type of node *S*, i.e., *CT*(*S*) returns *s* if *S* is *s-type* and returns *t* otherwise. On the other hand, ≠*CT*(*S*) returns *t* if *S* is *s-type* and returns *s*, otherwise. Function *decidePhase*() determines the current phase of routing by returning 0 to indicate that the subcube switch is the current required action, returning 1 to indicate that the current phase is the first phase, and returning 2 to indicate that the current phase is the second phase and is defined below.

$$decidePhase(p,S,Pd,curPhase,phase)=\begin{cases}2 & if\ (\text{curPhase}=1\ \wedge P^{phase}=Pd^{phase})\\ curPhase & otherwise\end{cases}$$

where v^{phase} returns the *s-subsequence* (*t-subsequence*) of the process id *v* if the routing phase, indicated by *phase*, is *s-phase* (*t-phase*).

After determining the current phase, routing in the current phase is carried out by identifying the neighbor to forward the message using functions *nextPosDummy*() and *nextPos*(). They both return the position in the *id* of process *p* to be flipped to obtain the *id* of the neighbor on the path leading to the destination neighbor *Pd*.

Function *nextPosDummy*() is used only for the dummy path in the first step of the subcube switch to determine the bit-flip position. We know that *knon-dummy* paths of the *k*+1 paths between *S* and *D* are routed initially in the subcube where source *S* exists in the first phase. In the first phase, the dummy path is routed in a subcube of type same as that of *S* but in a subcube different from the one containing *S* to ensure that the dummy path does not intersect with the other *k* paths. For that purpose, upon receipt of a message, the dummy neighbor *v* of *S* flips a bit to identify a neighbor in the same subcube containing *v* and then forwards the message. This neighbor of *v* is identified using function *nextPosDummy*(). Function *nextPosDummy* returns a bit position *j* such that $S^{phase}[j]=D^{phase}[j]$, if such a bit position does not exist, it returns a position such that it is not the first position in the *t-subsequence* or in the *s-subsequence*. Then, *v* forwards the message to its dummy neighbor which is of the same type as *S* but is in a different subcube than the one containing *S*. Now, the dummy path does not intersect with the other paths and the first phase of routing for the dummy path is started.

$$nextPosDummy(p,S,D)=\begin{cases} j & if\ \exists_{0<j<|P^{phase}|}S^{phase}[j]=D^{phase}[j]\\ 2 & otherwise\end{cases}$$

For the *s* and *t*-phases, the neighbor to forward the message is determined by function *nextPos*(). Function *nextPos*() returns the next position to be flipped within the *t-subsequence* if function *routPhase*() returns *t*, or within the *s-subsequence* if function *routPhase*() returns *s*. Function nextPos() returns the next most significant bit position after *f* where the current node id *p* and destination neighbor *Pd* differ, if exists, in their *s* or *t* sub-sequences depending on whether the current phase is the *s-phase* and *t-phase*. Otherwise, it returns the least significant bit position before *f*, if exists where the current node id *p* and destination neighbor *Pd* differ, if exists, in their *s* or *t* sub-sequences depending on whether the current phase is the *s-phase* and *t-phase*respectively. If no bit position where *p* and P_d differ exists in the current phase, it returns NULL. The function *nextPos*() is defined as

$$nextPos(p,Pd,phase,f)=\begin{cases} & j \quad if\ R_1\\ j & if\ \exists_{0<j<f^{\min(j)}}\wedge P^{phase}(j)\neq Pd^{phase}(j)\wedge \sqsubset R_1\\ null & otherwise\end{cases}$$

where $R_1\equiv\exists_{f<j<|p^{phase}|}\min(j)\wedge p^{phase}[j]\neq Pd^{phase}[j]$ and $v^{phase}[j]$ returns the bit value at position *j* in the subsequence specified by the routing *phase*.

VII. ALGORITHM

The distributed algorithm for routing over all node-disjoint paths in an exchanged hypercube is as follows:

Parameters:

M:	Message Sequence
m:	Message
S, *D*, *Pd*, *q*:	Process Id
f, *i*, *k*:	Integer
posToFlip:	Integer U NULL
curPhase:	{0, 1, 2}

begin

input(M, D) →
k := min(s, t) + 1;
for each i, $0 \leq i < k$
send(m_i, p, D, N(D, map(p, D, i)), *fPhase*(p, D, i), map(p, D, i)) to N(p, i);
rcv(m_i, S, D, Pd, curPhase, f) from q →

if $p = D$ *(message reached destination D)*
output(m_i); terminate ;
if $p = N(Pd, f) \neq D$ *(parameters Pd, f and D are inconsistent)*
output error; terminate ;
if $p = Pd$*(message reached a neighbor of destination)*
send(m_i, S, D, Pd, curPhase, f) to D; terminate;

curPhase = decidePhase(p,S,Pd,curPhase,routPhase(S,D,curPhase));
ifcurPhase = 0 *(subcube switch)*
posToFlip := nextPosDummy(p,D,⁻CT(S), f);
curPhase + +;
send(m_i, S, D,Pd,curPhase, posToFlip) to N(p,posToFlip);
else *(s-phase or t-phase routing)*
ifposToFlip := nextPos(p,Pd,routPhase(S,D, curPhase), q) ≠ NULL;
send(m_i, S,D, Pd, curPhase,posToFlip) to N(p,posToFlip);
terminate;
end

VIII. Correctness

The correctness of the algorithm is out of the scope of this version of the paper.

References

[1] R. L. Sharma, *Network Topology Optimization, The Art of and Science of Network Design*, Van Nostrand Reinhold, New York, NY, USA, 1990.
[2] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, Inc., New York, NY, USA, 1990.
[3] S. A. Plotkin, "Competitive routing of virtual circuits in ATM networks," *IEEE Journal of Selected Areas in Communications*, vol. 13, pp. 1128.
[4] D. Dolev, C. Dwork, O. Waarts, and M. Yung, "Perfectly secure message transmission," *J. ACM*, vol. 40, pp. 17, 1993.
[5] A. E. Kamal, "1+n protection in mesh networks using network coding over p-cycles," in *Proc. GLOBECOM*, 2006.
[6] X. Yang, G. M. Megson, S. Zhang, and X. Liu, "A solution to the three disjoint path problem on honeycomb meshes," *Parallel Processing Letters*, vol. 14, pp. 399-410, 2004.
[7] C.-N. Lai, "Optimal construction of all shortest node-disjoint paths in hypercubes with applications, parallel and distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1129-1134, 2012.
[8] P. K. K. Loh, W. J. Hsu, and Y. Pan, "The exchanged hypercube," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, pp. 866-874, 2005.
[9] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Transactions on Computers*, vol. 37, pp. 867-872, 1988.
[10] W.-K. Chen and M. F. M. Stallmann, "On embedding binary trees into hypercubes," *J. Parallel Distrib. Comput.*, vol. 24, pp. 132-138, 1995.
[11] J. Wu, "Adaptive fault-tolerant routing in cube-based multicomputers using safety vectors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, pp. 321-334, 1998.
[12] A. Bossard, K. Kaneko, and S. Peng, "Node-to-set disjoint-path routing in perfect hierarchical hypercubes," in *Proc. the International Conference on Computational Science*, vol. 4, pp. 442-451, 2011.
[13] S. Ziavras, "RH: A versatile family of reduced hypercube interconnection networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 11, pp. 1210-1220, 1994.
[14] O. Sinanoglu, M. H. Karaata, and B. AlBdaiwi, "An inherently stabilizing algorithm for node-to-node routing over all shortest node-disjoint paths in hypercube networks," *IEEE Transactions on Computers*, pp. 995-999, 2010.

Thamer Alsulaiman graduated from Kuwait University and is currently a PhD candidate in the University of Iowa. His research interests include distributed systems, networking, fault tolerant computing and self-stabilization.

Mehmet Hakan Karaata received his PhD degree in computer science in 1995 from the University of Iowa. He joined Bilkent University, Ankara, Turkey as an assistant professor in 1995. He is currently working as a Professor in the Department of Computer Engineering, Kuwait University. His research interests include mobile computing, distributed systems, fault tolerant computing and self-stabilization.