Fault Tolerance in Hypercubes

Shobana Balakrishnan, Füsun Özgüner, and Baback A. Izadi

Department of Electrical Engineering, The Ohio State University, Columbus, OH 43210, USA

Abstract: This paper describes different schemes for tolerating faults in hypercube multiprocessors. A study of hypercube algorithms reveals that in many cases, the computations that require local communication are mapped onto topologies such as meshes or rings and the hypercube topology is used for global data communication. Therefore, a faulty hypercube needs to be reconfigured to perform both local and global communication as required by the algorithm, effectively and with minimal performance degradation. Two general approaches can be identified. The first approach looks into ways of utilizing the healthy processors and links of a hypercube with faulty nodes/links, for embedding topologies such as lower dimensional hypercubes, rings, meshes and trees for performing communication. The second approach makes use of hardware redundancy in the form of spare nodes and/or links and usually requires modifications in the communication hardware. Augmented hypercubes and spare allocation schemes are described.

Keywords: Hypercube, fault tolerance, embeddings, global communication, spare allocation, reconfiguration

1 Introduction

The area of reconfigurability in the presence of faults is becoming increasingly important with the emergence of massively parallel architectures. Therefore, fault tolerance is an important issue that needs to be addressed in the design of node architectures, communication hardware and software design as well as parallel algorithm development. A d–dimensional hypercube multicomputer consists of $n = 2^d$ processors (nodes) interconnected as a Boolean d-cube, with each processor having only local memory. Inter–processor communication is done by explicit message passing.

Each processor in a d–cube can be represented by a ***d–tuple*** $(b_{d-1} \cdots b_i \cdots b_0)$ where $b_i \in \{0, 1\}$, and a subcube in a d-cube can be represented by a ***d–tuple*** $\{0, 1, \mathrm{x}\}^d$. Coordinate values "0" and "1" can be referred to as *bound* coordinates and "x" as *free*. A $(d - k)$–dimensional subcube ($(d - k)$–subcube) in a d–cube is represented by a ***d–tuple*** with *k bound* coordinates

and *d-k free* coordinates.

Two general approaches can be identified for fault tolerance in hypercube multiprocessors. The first approach looks into ways of utilizing the healthy processors and links of a hypercube with faulty nodes/links, to identify embedded topologies such as lower dimensional hypercubes (subcubes), meshes, etc. required by the computations. With this approach, some performance degradation is expected, however special hardware design for fault tolerance is not necessary. Embeddings are discussed in Section 2 and an algorithm for finding fault-free subcubes is described in Section 3.

A distributed fault diagnosis algorithm for identifying the set of faulty processors and links, when the number of faulty processors and links $r \leq d$ is given in [1]. Once the faulty elements are identified, the multiprocessor and the distributed algorithm running on the multiprocessor[2] must be reconfigured to run on the available processors. A faulty hypercube needs to be reconfigured to perform both local and global communication as required by the algorithm, effectively and with minimal performance degradation. A study of hypercube algorithms reveals that in many cases, the computations that require local communication are mapped onto topologies such as meshes or rings (grid problems for example [3]) and the hypercube topology is used for global data communication. Therefore, both local and global communication schemes must be considered in designing algorithms for faulty hypercubes. The problem of communication in a faulty hypercube has been addressed by many researchers. A survey of some of these schemes as well as new results will be presented in Section 4.

The second approach to fault tolerance makes use of hardware redundancy in the form of spare nodes and/or links and usually requires modifications in the communication hardware. Although a d-dimensional hypercube with faulty nodes still contains a number of lower dimensional subcubes and therefore can be used to run hypercube algorithms without any modification [4, 5, 6, 7], one faulty node reduces the dimension of the largest fault-free subcube to $(d-1)$ and 2 faulty nodes can reduce it to $(d-2)$. This has motivated researchers to investigate hardware redundancy and spare allocation schemes [8, 9, 10, 7, 11, 12, 13, 14, 15]. Augmented hypercubes and spare allocation schemes will be described in Section 5.

2 Embeddings

Embedding problems are concerned with finding mappings between two graphs that preserve certain topological properties. Let $G_a = (V_a, E_a)$ be an application graph and $G_h = (V_h, E_h)$ be the host graph that represents the interconnection topology of the parallel computer. An embedding ϕ of a graph G_a into a graph G_h is a function from V_a to V_h [16]. ϕ is an isomorphic embedding *iff* $G_a \subseteq G_h$ and for all $u, v \in V_a,\ d_a(u,v) = d_h(\phi(u), \phi(v))$ where $d(u,v)$ is the distance between vertices u and v. The *dilation* of ϕ is $Max(d(\phi(u), \phi(v)))$ for all $u, v \in V_a$. Note that an isomorphic embedding is a unit dilation embedding.

Previous research has shown that a number of useful graphs can be embedded isomorphically into healthy hypercubes. A graph G is *cubical* if there exists an isomorphic embedding of G in the d-cube graph. The focus of the research has been to determine the largest size of the graph G that can be embedded in a d-cube. For example, in [17] and [18] it is shown that trees are cubical, and that a $(d-1)$–height tree can be embedded in a d-cube isomorphically. In cases where adjacency cannot always be preserved, embeddings of dilation greater than one have been determined [19, 20, 21, 22].

In this section, we present an overview of algorithms for embedding trees, rings and meshes in faulty hypercubes. We discuss embeddings that handle node failures only, unless otherwise stated. The direction of the research in fault–tolerant embeddings has followed two different paths depending on the motivating problem.

The first approach [19, 20] is to find embedding functions that primarily minimize the cost of reconfiguration; *i.e.*, the number of node state changes during reconfiguration is minimized. Generally, such research has focused on distributed algorithms so that the neighbors of the faulty node can detect the faulty node and locally decide how to remap the faulty node to a healthy node with minimum reconfiguration cost. Remapping in this way may not always preserve adjacency, thus increasing the communication cost. Furthermore, the number of nodes utilized in the embedding is not always maximized. Such algorithms also do not maximize the number of faults that can be tolerated and exist for single or double fault cases only.

The second direction taken by researchers [23, 24] in fault–tolerant embeddings has been to preserve the adjacency of nodes; *i.e.*, produce an isomorphic embedding. Hence, for unit dilation embeddings, the focus has been to determine the maximum size of the embedding so that fewest number of healthy nodes are wasted (unused). Bounds on the maximum number of faults that can be tolerated are also obtained.

2.1 Rings

In this section we review the approaches taken in [20] and [23]. We elaborate on the idea used in [23] since it handles multiple faults and creates the fault–tolerant embedding at run time in a distributed manner.

The key idea in [20] is to embed the ring so that unused (healthy) nodes are close to all nodes in the embedding, whereby, an unused node can replace a faulty node in the embedding easily with a few reconfiguration steps. Fig. 1 (dark lines) illustrates a ring of length eight embedded in a 4-cube. In general, the algorithm utilizes a small percentage of the nodes where the maximum is 50%. It can be seen from Fig. 1 that such an embedding can tolerate two faults and the reconfiguration algorithm requires two state changes *i.e.*, nodes 1010 and 1000 are added in place of the faulty nodes 0000 and 0001 respectively.

Embedding a ring in a d-cube so that the size of the ring is maximized is the approach taken

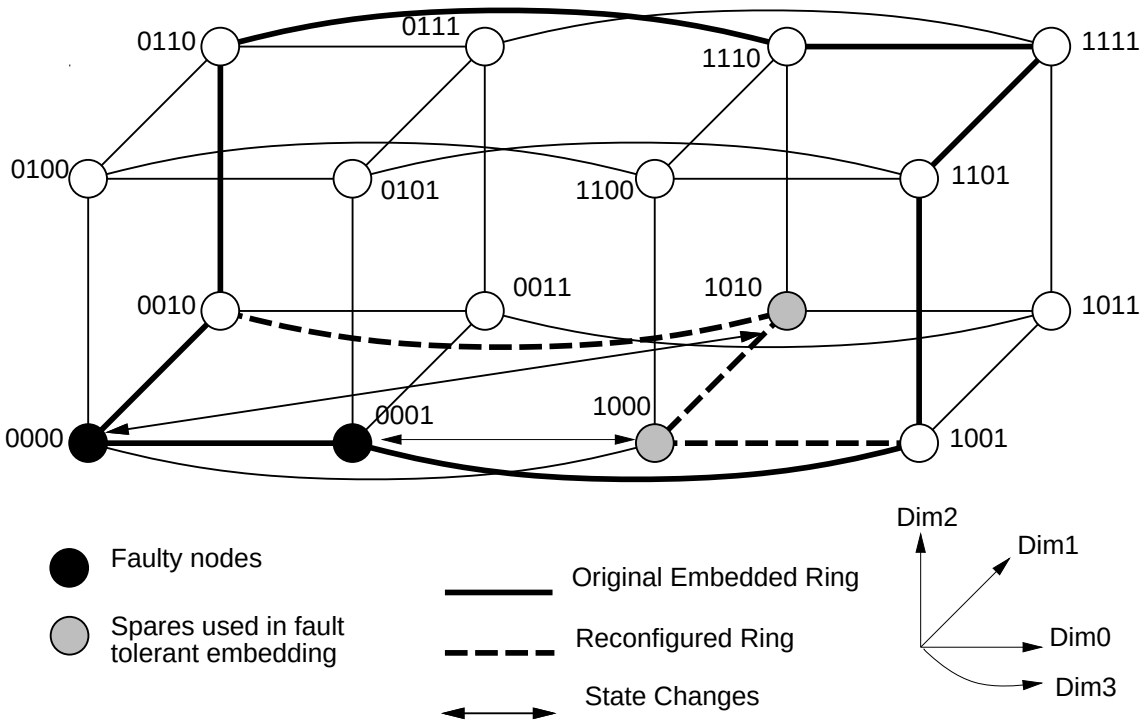

Figure 1: Ring embedding minimizing reconfiguration cost

by [23]. The authors devise a distributed algorithm that embeds a ring of size $2^d - 2f$ in a d-cube where the number of faults $f \leq \lfloor \frac{d+1}{2} \rfloor$. They use cyclic Gray codes to embed the ring as is done in healthy hypercubes [18] and vary this *basis ring* to skip over the faulty nodes. Fig. 2 shows a 4-cube with the basis ring embedded (shown by tracing the number in the angle brackets *i.e.*, $<>$ starting at node 0000). The embedding dimension sequence for the basis ring in a 4-cube denoted as $DM_0 = \{0, 1, 2, 3\}$ is $(0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 3)$ where following this dimension sequence from the source node results in the basis ring. Let us also denote the *rank* of the processor as the location of the processor in the embedded basis ring. The number in the angle bracket in Fig. 2 represents the rank of the processor. For example, the source has rank 1, and for the embedding sequence $\{0, 1, 2, 3\}$ the neighbor of the source along dimension 0 has rank 2 while the neighbor of the source along dimension 3 has rank 16. The first dimension in DM_0 is the alternating dimension (*i.e.*, the dimension traversed most frequently), and is denoted as d_f ($d_f = 0$ in this example). When a faulty node is encountered along the embedded ring, the algorithm in [23] executes one of the following two cases. In the first case, if the dimension along which the faulty node is reached is the most frequently occurring dimension d_f(*i.e.*, dimension 0 in the example), then the dimension sub-sequence $[d_f, d', d_f]$ is replaced by $[d']$ where $d' \neq d_f$. For example, as in Fig. 2 (a), if node 0111 is faulty, then at node 0110 (which is the fifth processor in the ring) the next dimension along the embedding sequence is changed from 0 to 1 *i.e.*, the dimension sub-sequence $[0, 1, 0]$ is replaced by $[1]$. In the second case, if the dimension along which the faulty processor is reached is not dimension d_f but some other dimension d', then the dimension sub-sequence $[d_f, d', d_f]$ is replaced by d', *i.e.*, the ring

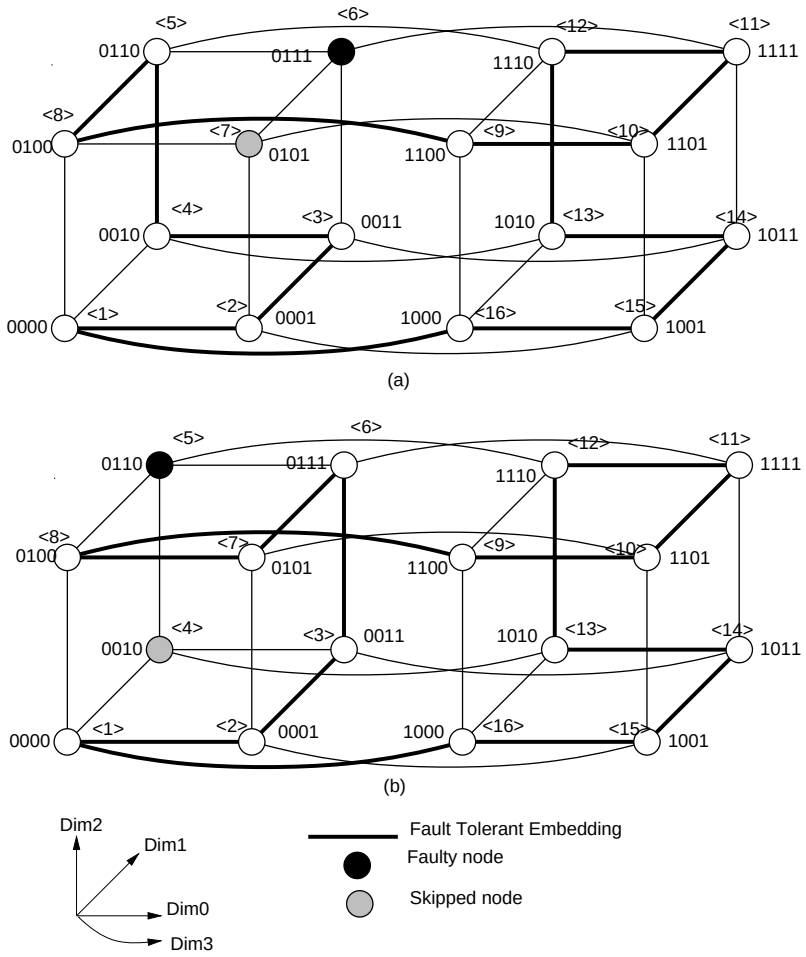

Figure 2: Ring embedding maximizing size of ring: (a) Faulty processor reached along dimension 0 (b) Faulty processor reached along dimension 2

so far constructed is *backtracked* along d_f and the ring construction is continued by the previous processor in the embedding. For example, as in Fig. 2 (b), if processor 0110 is faulty (which is the processor accessed along dimension 2), the algorithm backtracks to processor 0011 and replaces the dimension sub-sequence $[0, 2, 0]$ by $[2]$ with the next dimension pointer pointing to dimension 1.

The algorithm is guaranteed to find a ring embedding (where for each faulty processor, a healthy processor is removed from the original embedding) provided that the number of faults does not exceed $\lfloor \frac{d+1}{2} \rfloor$. In the first case, when the faulty processor is reached along d_f, the faulty processor has an even rank in the embedding sequence. The algorithm skips the faulty processor and the processor after it in the fault–tolerant embedding. In the second case, when the faulty processor has an odd rank in the original embedding sequence, the algorithm skips the faulty processor and the processor before it in the fault–tolerant embedding. To tolerate multiple faults the algorithm handles fault distributions where (1) no two faults are distance two apart *and* (2) if two faults are distance one apart then the first faulty processor encountered in the original basis

ring must have an even rank. The authors fix the number of faults f, and permute the dimension sequence of the basis ring $2f-1$ times, by defining $2f-1$ embedding dimension sequences $DM_0, DM_1, ..., DM_{2f-2}$ where DM_i is DM_0 with dimension 0 and dimension i swapped. The number of faults is fixed to not exceed $\lfloor \frac{d+1}{2} \rfloor$ since the number of permutations cannot exceed d. It is shown that there exists at least one dimension sequence for which the basis ring satisfies the above two conditions and is guaranteed to find a fault-tolerant embedding of size $\geq 2^d - 2f$.

2.2 Meshes

In a faulty hypercube, an $m_1 \times m_2$ mesh can be obtained by removing faulty rows and/or columns in a $2^{d_1} \times 2^{d_2}$ mesh, where $d_1 \geq \lceil log_2 m_1 \rceil$, $d_2 \geq \lceil log_2 m_2 \rceil$, and $(d_1 + d_2) \leq d$, and finding a Gray code ordering of the m_1 healthy rows and m_2 healthy columns [24]. An example of embedding a 5×5 mesh in a faulty 6-*cube* is illustrated in Fig. 3. An initial embedding of an 8×8 mesh shown in Fig. 3 is done by generating a 2-D Gray code for the coordinates $b_5 b_4 b_3$ and $b_2 b_1 b_0$. A 5×5 mesh is obtained by removing the faulty rows and/or columns in Fig. 3 and then by rearranging the rows and columns in Gray code ordering. For this example, the rows would be arranged as [101, 100, 000, 010, 011] and the columns as [001, 000, 100, 110, 111]. However, the search for a Gray code ordering of healthy rows and columns is computationally complex. Pasting of the healthy rows/columns for constructing either side of the mesh is equivalent to embedding a ring in a faulty d_1-subcube or a faulty d_2-subcube respectively and can be constructed using the ring embedding techniques so that a faulty node is skipped along with a node adjacent to it. In the worst case this results in two rows or columns of nodes being skipped for every fault, resulting in an embedded mesh of size $(2^{d_1} - 2x) \times (2^{d_2} - 2y)$, where $x + y = f$.

However, if the hypercube has *Direct-Connect Capability* [25] and assuming that only processors can fail, then a linear array of any length m can be formed by first forming a linear array of length 2^d in a d-cube using a Gray code sequence and then connecting through the *Direct-Connect Modules* (DCM's) of the faulty nodes. An $m_1 \times m_2$ mesh can be formed similarly by pasting the healthy rows and columns through the DCM's of faulty rows/columns (Fig. 3). In routing messages, if the *e-cube* routing algorithm [25, 26] is used (dimension order routing starting with dimension 0), communication from the nodes of subcube $\mathrm{xxx}001$ (*column 001*) to the nodes of subcube $\mathrm{xxx}110$ (*column 110*) will go through the DCM's of subcube $\mathrm{xxx}000$ and subcube $\mathrm{xxx}010$; on the other hand, communication from the nodes of subcube $\mathrm{xxx}110$ to the nodes of subcube $\mathrm{xxx}001$ will go through the DCM's of subcube $\mathrm{xxx}111$ and subcube $\mathrm{xxx}101$. Thus e-cube communication is not suitable for two reasons: one is that routing is through a different set of processors in each direction. The more important reason is that a link connecting 2 processors in the mesh (such as $\mathrm{xxx}000$ and $\mathrm{xxx}001$) may be used, which is also used for local (nearest neighbor) communication between those 2 processors. In order to guarantee regular and uninterrupted local communication, messages between two processors in

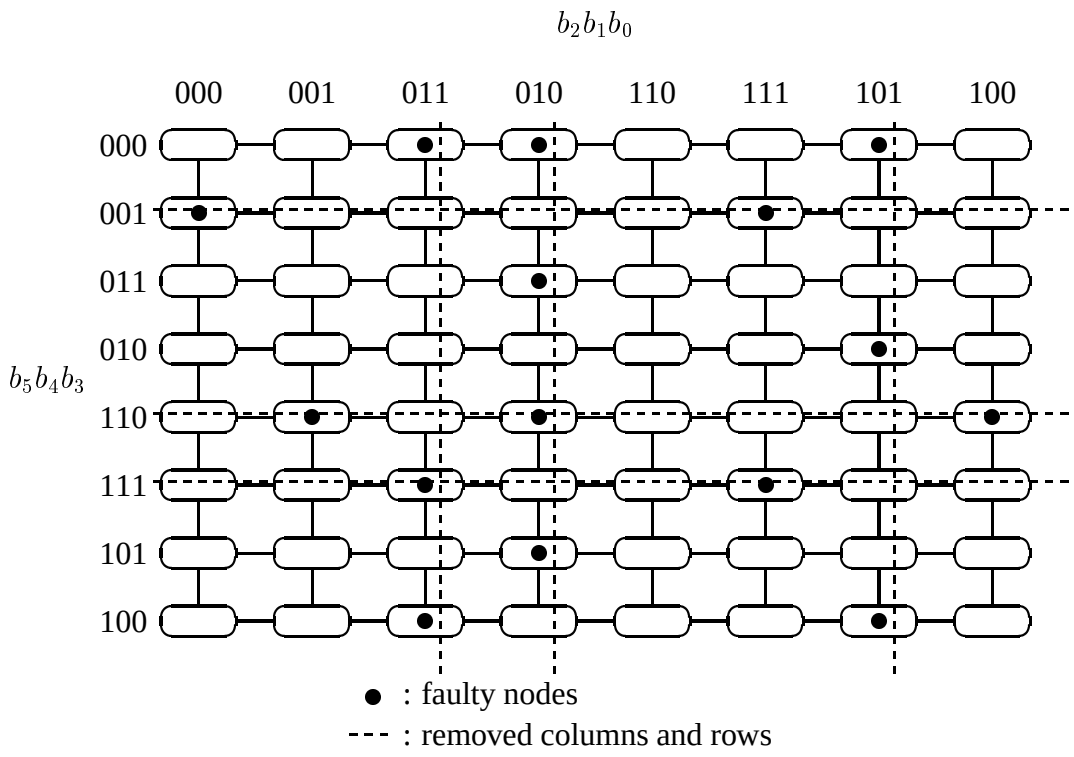

Figure 3: A 5×5 mesh embedded in a 6-cube

pasted rows/columns must go through the DCM's of the deleted rows/columns. In the example shown in Fig. 3, messages between the nodes of subcube $\mathrm{xxx001}$ (*column 001*) and the nodes of subcube $\mathrm{xxx110}$ (*column 110*) should go through the DCM's of subcube $\mathrm{xxx011}$ and subcube $\mathrm{xxx010}$ in both directions. Thus the DCM must be designed to route messages in both directions in accordance with the coordinate sequence used in defining each side of the mesh.

In a specific Karnaugh-map illustration of a $2^{d_1} \times 2^{d_2}$ mesh, the group of nodes in a column or a row form a d_1-subcube or a d_2-subcube, and are represented by a unique d-*tuple*$\{0, 1, \mathrm{x}\}^d$. In algorithm *MESHDCM* [24], we ignore all the *free* coordinates "x" and consider the "bound" coordinates only in deleting faulty rows/columns and thus the group of nodes in a row or a column are identified by a unique d_1-*tuple* or a unique d_2-*tuple* respectively. The faulty and healthy nodes of a row/column have the same representation when they are indicated by considering the "bound" coordinates only. Therefore, deleting any one faulty node in a row/column is equivalent to deleting the entire row/column. *MESHDCM* given below, first chooses d_1 coordinates among the d coordinates to construct the m_1 side of the $m_1 \times m_2$ mesh, where $d_1 = \lceil log_2 m_1 \rceil$, and then constructs the m_2 side of the mesh by using the remaining $(d\text{-}d_1)$ coordinates.

Algorithm *MESHDCM:*

Step 1: If $(d - d_1) < d_2$ or $(2^d - f) < (m_1 \times m_2)$, then exit.

Step 2: Calculate the number of extra rows $N_e = (2^{d_1} - m_1)$, that can be deleted for forming the m_1 side of the mesh.

Step 3: Choose d_1 coordinates among the d coordinates to construct the m_1 side of the mesh. Then construct the list F_{d_1} by representing the f faulty elements in the fault list F by the d_1 coordinates only ($\mid F_{d_1} \mid \leq 2^{d_1}$), and deleting identical representations of the faulty nodes in the d_1 coordinates.

Step 4: If $\mid F_{d_1} \mid > N_e$ then construct F_{m_1} by choosing N_e elements from the fault list F_{d_1}. Else let $N_e = \mid F_{d_1} \mid$ and $F_{m_1} = F_{d_1}$, and choose any m_2 columns and exit. The N_e elements in F_{m_1} correspond to the rows to be deleted represented in the d_1 coordinates.

Step 5: Construct F_{m_2}, the list for constructing the m_2 side, by deleting from F the faulty nodes in the rows in F_{m_1}, then represent the faulty elements by the d_2 coordinates only deleting identical representations. Calculate the number of healthy columns $N_c = 2^{d_2} - \mid F_{m_2} \mid$. If $N_c < m_2$, then go to Step 6, else exit.

Step 6: Check if all the possible choices of N_e nodes have been used exhaustively in Step 4. If yes, then go to Step 7, else go to Step 4 for the next trial.

Step 7: Check if all the combinations of choosing d_1 coordinates among the d coordinates are examined exhaustively. If yes, then exit, else go to Step 3 for the next trial.

The complexity of Algorithm *MESHDCM* is analyzed as follows. In the worst case, $\binom{d}{d_1}$ combinations of coordinates will be tried to construct the m_1 side of the mesh and for each choice, there are $\binom{|F_{d_1}|}{N_e}$ ways of deleting rows. Step 3 has a complexity of $flog_2f$ and Step 5 has a complexity of $(fN_e + flog_2f)$. Therefore, the time complexity is $O(\binom{d}{d_1}[flog_2f + \binom{|F_{d_1}|}{N_e}(flog_2f + fN_e)])$. If $f < d$, ($N_e < d$) a pessimistic worst case complexity of $O(P^2d^2)$ can be derived since $\binom{d}{d_1} < P$, and $\binom{|F_{d_1}|}{N_e} < P$, where $P = 2^d$.

An example of embedding a 3×3 mesh in a 4-cube with 3 faulty nodes (f_1=0001, f_2=0101, and f_3=1100) by Algorithm *MESHDCM* is shown in Fig. 4. Since d=4, m_1=3, and m_2=3, $(d - \lceil log_2m_1 \rceil) = 2$, $\lceil log_2m_2 \rceil = 2$, and $(2^d - f) = 13 > (m_1 \times m_2) = 9$, the mesh embedding is possible. Then we have N_e = (2^{d_1}-m_1) = 1. Since $d_1 = \lceil log_2m_1 \rceil$ = 2, there are $\binom{4}{2}$ ways of choosing 2 coordinates among 4 coordinates. As shown in Fig. 4, b_3b_2 is selected first and the faulty nodes represented in b_3b_2(elements of F_{d_1}) are $F_1 = 00$, $F_2 = 01$, F_3 = 11. In trial 1, F_1 is deleted and F_{m_2} is $\{01, 00\}$. Since N_c = 2 $< m_2$, trial 2 is required. In trial 2, F_2 is deleted and F_{m_2} is $\{01, 00\}$. Since N_c = 2 $< m_2$, trial 3 is required. In trial 3, F_3 is deleted, F_{m_1} is $\{11\}$, F_{m_2} is $\{01\}$ and N_c = 3 = m_2.

2.3 Binary Trees

Another topology that is often used in algorithms is a binary tree. Two significant results regarding tree embeddings have emerged. It was shown in [21] that a tree of height d, T_d can be

Illustration of Trial 1:

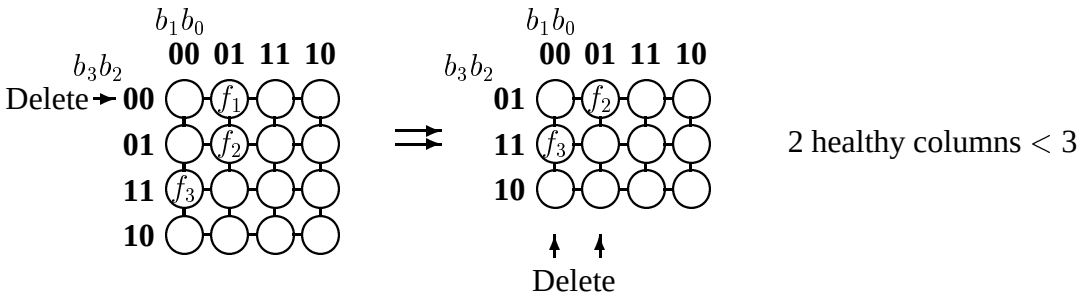

Illustration of Trial 2:

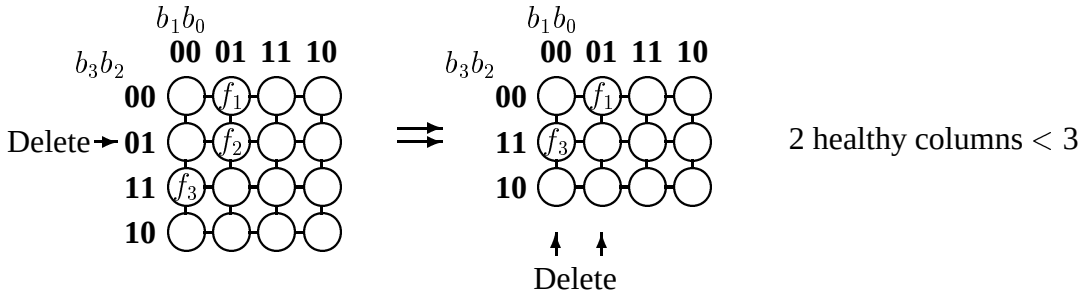

Illustration of Trial 3:

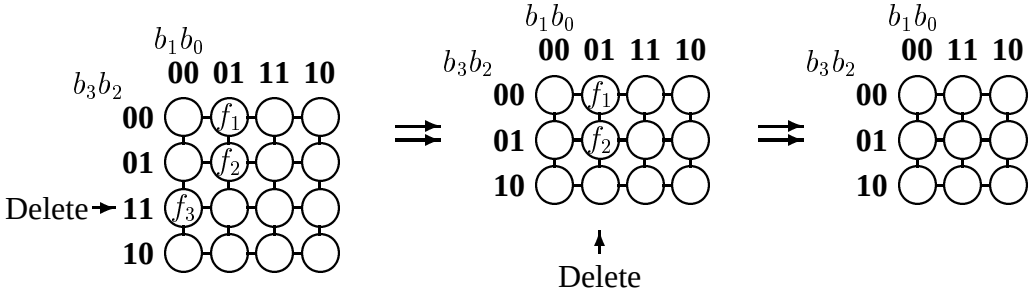

Figure 4: Illustration of a 3x3 mesh embedding using algorithm MESHDCM

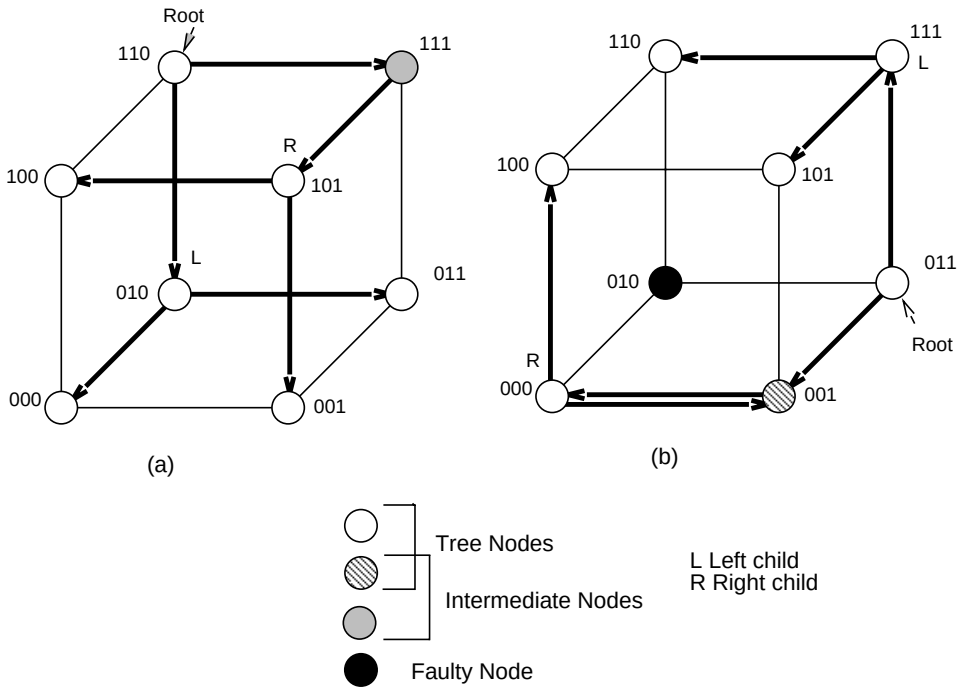

Figure 5: Tree embedding maximizing the size of the embedding (a) T_3 with dilation 2 - basis embedding (b) T_3 with dilation 2 - fault-tolerant embedding

embedded in a d-cube with maximum dilation 2, *i.e.*, the adjacency of nodes in a tree are not preserved. Here, the motivation is to utilize the maximum number of nodes thus trading off the increased dilation for a maximum embedding size. The other approach taken in [17, 18] was to show that a binary tree of height $d-1$, T_{d-1}, can be embedded in a d-cube with unit dilation. Here the trade-offs are just the reverse of the previous approach, *i.e.*, preserving adjacency results in not maximizing the number of nodes used.

Fault–tolerant embeddings have been considered using one of the above two approaches as the basis embedding. The factors considered in the fault–tolerant embedding are: (1) the number of faults tolerated, (2) the dilation of the fault–tolerant embedding, (3) the reconfiguration cost *i.e.*, the number of nodes in the new embedded graph that do not have the same state as in the basis embedding and (4) the size of the embedding.

In [21] the basis embedding is T_d where the maximum dilation is 2 as shown in Fig. 5 (a) by the dark lines. The approach as seen in Fig. 5 (b) is to use the intermediate node *i.e.*, node 111 (in the multi hop edge of the tree) to replace a faulty node (node 010) and utilize another tree node (node 001) in the basis embedding as an intermediate node as well as a tree node in the fault-tolerant embedding. Here the objective is to minimize the dilation of the fault–tolerant embedding (*i.e.*, keep it constant at 2) and not degrade the size of the embedding (*i.e.*, embed T_d). Only one fault is tolerated by this scheme. Note also that the new fault–tolerant embedding results in the states of all the nodes being changed as shown in Fig. 5 (b). For example, the root of the tree is mapped to node 011 instead of 110, the left child to 111 instead of 010, the right

child to 000 (via 001) instead of 101 (via 111) and so on. Improvements to this result have been made in [20]. Here, a fault–tolerant embedding with T_{d-1} as the basis embedding is used so that the dilation is minimized (*i.e.*, kept constant at 1) and 2 faults are handled. This approach has the same penalty as before *i.e.*, a reconfiguration cost of $2^{d-1}-1$. Furthermore, the height of the embedding is $d-1$ due to the approach taken for the basis embedding.

The other approach taken is to minimize the reconfiguration cost [27, 22]. Here the basis embedding has unit dilation but the fault–tolerant embedding has a dilation of 2 which is the penalty paid to obtain a reconfiguration cost of 1. In [22] the reconfiguration algorithm involves two state changes if the faulty node is at height 2 in the basis embedding. Both these algorithms use a mirror image of the faulty node along some dimension as the spare to replace the faulty node. The authors in [22] use a distributed algorithm where the faulty node is remapped and only its parent and children are aware of the change. The authors also extend their algorithm to tolerate multiple faults. Here they use a form of backtracking where if a node cannot embed its children, it declares itself faulty to its parent who must handle the reconfiguration. In the worst case, the algorithm will backtrack up to the root and the embedding will fail.

The results discussed so far have not considered maximizing the number of faults that can be tolerated. In [28] the authors show that in a d-cube with $d-1-\lceil log_2 d \rceil$ node and/or link faults, there exists a $d-1$ tree which avoids the faults. Then the fault avoiding $(d-1)$ tree is constructed in two steps. First, a fault-free tree T' is constructed in which each node has exactly two or no children and the leaves of T' are at level $d-k$ or level $d-k-1$, where the level of a node is the height of the node from the root and $k=\lceil log_2 d \rceil + 2$. Second, fault avoiding $(k-1)$ trees (*i.e.*, of height $k-1$) are constructed to "attach" onto the leaves of T' at level $d-k$ and fault avoiding (k) trees to attach onto the leaves at level $d-k-1$. It is shown that a k-subcube can be associated with the leaves at level $d-k$ and a $(k+1)$-subcube with the leaves at level $d-k-1$. These subcubes which partition the d-cube have the following properties. Exactly one of the leaves of T' and at most *two* internal nodes of T' belong to each of the $k-$cubes and each of the k-cubes has at most *one* fault. Exactly one of the leaves of T' and at most *one* internal node of T' belong to each of the $(k+1)-$cubes and each of the $(k+1)$-cubes has at most *two* faults. Within such subcubes fault avoiding $(k-1)$-trees and k-trees can be constructed.

3 Finding Fault-Free Subcubes

Parallel algorithms for hypercubes can be formulated with the dimension d of the hypercube as a parameter of the algorithm[5] and therefore can be run on a fault-free subcube of the faulty hypercube. A simple distributed procedure to find the maximum dimension m_d of a fault-free subcube is given in [5]. However, as indicated in [5], this procedure does not always find m_d and furthermore, it does not construct the set of fault–free m_d–subcubes. An algorithm which always finds m_d and also the complete set of fault-free m_d-subcubes is presented in [4]. The

results are briefly explained below. The algorithm is formulated to run on a single processor which would typically be the host or the resource manager in a commercial hypercube system. Note that there are $2^k\binom{d}{k}$ different $(d-k)$–subcubes and a total number of $\sum_{i=1}^{d} 2^i\binom{d}{i} = 3^d - 1$ different subcubes in a d–hypercube.

A commutative and associative intersection operation $\mathbf{I_d} = \{\mathbf{I}\}^d$ on the faulty processors is defined as follows where **I** denotes the intersection of individual coordinates; $0\mathbf{I}0 = 0$, $1\mathbf{I}1 = 1$, $0\mathbf{I}1 = q$, and $q\mathbf{I}0 = q\mathbf{I}1 = q\mathbf{I}q = q$. The *bound* coordinates ("0"s and "1"s) in the intersection of all faulty processors indicate the *fixed* coordinates among them. Hence, the number of *fixed* coordinates, n_f, gives the number of $(d-1)$-dimensional fault-free subcubes. Finding $(d-k)$–dimensional fault-free subcubes for $k > 1$ is more complex and will be described later in this section. The algorithm given in [4] is based on using the *Inclusion-Exclusion*[29] principle to count the number of subcubes of a given dimension $(d-k)$, that can be formed in the presence of faulty processors and links.

Lemma 1[4]: The number of $(d-k)$–subcubes destroyed by a faulty processor is $D_{d-k} = \binom{d}{k}$.

Let $S_{F_i}^{d-k}$ denote the set of $(d-k)$–subcubes that faulty processor F_i belongs to. Subcubes jointly destroyed by ℓ faulty processors $F_{i_1}, F_{i_2}, \ldots, F_{i_\ell}$ or subcubes in $S_{F_{i_1}}^{d-k} \cap S_{F_{i_2}}^{d-k} \cap \cdots S_{F_{i_\ell}}^{d-k}$ can be counted without explicit enumeration of subcubes in each set by using the following lemma.

Lemma 2[4]: The number of $(d-k)$–dimensional subcubes jointly destroyed by ℓ faulty processors $F_{i_1}, F_{i_2}, \ldots, F_{i_\ell}$ is

$$K_{d-k}(F_{i_1}, F_{i_2}, \ldots, F_{i_\ell}) \;=\; \binom{n_f}{k} \tag{1}$$

where n_f is the number of fixed coordinates in $I_{i_1 i_2 \cdots i_\ell} = F_{i_1}\mathbf{I_d}F_{i_2}\mathbf{I_d}\cdots\mathbf{I_d}F_{i_\ell}$.

For example, in a 5-cube, if the faulty processors are $F_1 = (00011)$ and $F_2 = (01111)$, then $I_{12} = F_1\mathbf{I_d}F_2 = (0qq11)$. For $k = 2$, $d - k = 3$ the 3–subcubes (xxx11), (0xxx1) and (0xx1x) are destroyed by both F_1 and F_2. Note that F_1 and F_2 each destroy ten 3-cubes.

In counting the number of *distinct* $(d-k)$–subcubes destroyed by faulty processors $F_1, F_2, ..., F_r$, the *common* $(d-k)$–subcubes destroyed should not be *included* more than once in the count and should be *excluded* by using the *Inclusion-Exclusion*[29] principle of counting given below.

Theorem 1[29]: **Principle of Inclusion-Exclusion**. If N is the number of elements in a set S, the number of elements of S not having any of the properties $p_1, p_2, ..., p_r$ is given as:

$$\begin{aligned} N(p_1' p_2' \cdots p_r') \;=\; & N - \sum_{i=1}^{r} N(p_i) + \sum_{i \neq j} N(p_i p_j) - \sum_{i,j,k} N(p_i p_j p_k) \\ & + \cdots + (-1)^r N(p_1 p_2 \cdots p_r) \end{aligned} \tag{2}$$

Here, $N(p_i)$ denotes the number of objects having property p_i, $N(p_i p_j)$ denotes the number of objects having both properties p_i and p_j, and so on. Objects having the same property p_i

are elements of the set S_i. The second summation on the right hand side of the equation is over all pairs of sets $S_iS_j(i \neq j)$ and therefore enumerates the number of elements in pairwise intersections of sets S_i and S_j, the third summation is over triples $S_iS_jS_k$ and finally the last term enumerates the number of elements in the intersection of sets $S_1 \cdots S_r$.

This principle can be applied to counting the number of fault-free or available subcubes of a given dimension $(d-k)$, G_{d-k}, as described by the following theorem.

Theorem 2[4]: The number of fault-free (available) $(d-k)$–subcubes, G_{d-k}, in the presence of r faulty processors $F_1, F_2, ...F_r$ is

$$\begin{aligned} G_{d-k} &= 2^k \binom{d}{k} - r\binom{d}{k} + \sum_{i \neq j} K_{d-k}(F_iF_j) - \sum_{i,j,k} K_{d-k}(F_iF_jF_k) \\ &\quad + \cdots + (-1)^r K_{d-k}(F_1F_2 \cdots F_r) \end{aligned} \tag{3}$$

The steps of the algorithm for finding m_d are given as:

1. *Construct the intersection $I_{12\cdots r}$ and compute n_f for this intersection. If $n_f \geq 1$ then exit with $m_d = d-1$ and $G_{d-1} = n_f$.*
2. *Construct all pairwise intersections $I_{i_1 i_2}$ and compute n_f for these intersections.*
3. *Construct intersections $(I_{i_1 \cdots i_{\ell-1}})\mathbf{I_d}F_\ell$, $2 < \ell < r$, for which $n_f(I_{i_1 \cdots i_{\ell-1}}) \geq 2$ and $n_f(I_{i_j i_\ell}) \geq 2$ for all $i_j \in \{i_1, i_2, \ldots, i_{\ell-1}\}$, and compute n_f for these intersections.*
4. **for** $k = 2, 3, \ldots$ **do**
   (a) *compute $K_{d-k}(F_{i_1}, F_{i_2}, \cdots, F_{i_\ell})$ from equation(1), for $1 < \ell < r$.*
   (b) *compute G_{d-k} from equation 3.*
   (c) *exit the loop if $G_{d-k} \neq 0$ with $m_d = d-k$* and $G_{m_d} = G_{d-k}$.

The complexity of the algorithm is $O(k2^r)$ which occurs only when all of the intersections have to be constructed and $n_f \geq k$ for each intersection. With the assumption that $r \leq d$[1], the complexity can be expressed as $O(k2^d)$.

Once $m_d = d-k$ and G_{m_d} are found, an m_d-dimensional subcube can be found as follows. There are no $(d-k)$–dimensional fault-free subcubes, if all the 2^k combinations of $\{0,1\}^k$ are exhaustively covered at each $\binom{d}{k}$ different k-coordinate position combinations by the set of faulty processors. If any one of the 2^k combinations of $\{0,1\}^k$, at any one of the $\binom{d}{k}$ k-coordinate position combinations, is found to be missing in the list of faulty processors, then the *d–tuple* constructed by assigning that missing *k–tuple* combination to those particular k coordinate positions and also assigning the remaining $d-k$ coordinates as *free* ("x"), defines a $(d-k)$–dimensional fault-free subcube. The search for missing combination(s) for each faulty

element can be avoided by encoding each *k–tuple* at a k-coordinate position combination of faulty elements with a 1-out-of-2^k code and then simply using the logic *OR* operation on these encoded 2^k*–tuples*.

Consider the following example:

	b_3	b_2	b_1	b_0
F_1 :	0	0	0	0
F_2 :	0	1	0	0
F_3 :	0	1	1	0
F_4 :	1	0	0	1

Equation 3 is used to find the number of fault-free 2-cubes as follows:
For k=2 (d-k=2);
$I_{12} = (0q00)\ n_f = 3,\ \ I_{13} = (0qq0)\ n_f = 2,\ \ I_{23} = (01q0)\ n_f = 3,$
$I_{14} = (q00q)\ n_f = 2,\ \ I_{24} = (qq0q)\ n_f = 1,\ \ I_{34} = (qqqq)\ n_f = 0,$
$I_{123} = (0qq0)\ n_f = 2,\ \ \ I_{124} = (qq0q)\ n_f = 1,$
$I_{134} = I_{234} = I_{1234} = (qqqq)\ n_f = 0,$

$\sum_{i \neq j} K_{d-k}(F_iF_j) = \binom{3}{2} + \binom{2}{2} + \binom{3}{2} + \binom{2}{2} = 8$

$\sum_{i,j,k} K_{d-k}(F_iF_jF_k) = \binom{2}{2} = 1$

$G_2 = 2^2\binom{4}{2} - 4\binom{4}{2} + 8 - 1 = 7$. Since $G_2 \neq 0$, the maximum dimension $m_4 = 2$.

b_1b_0=11 is one of the missing combinations and therefore the 2-cube $\mathrm{xx11}$ is found as one of the fault-free 2-cubes.

The upper bound for the complexity of finding the missing combination is $O\left((rk + 2^k)\binom{d}{k}\right)$, when the number of faults r is not restricted. A simpler bound is derived in [4] for $r \leq d$.

4 Communication in Faulty Hypercubes

4.1 Global Sum/Global Broadcast Algorithms for Healthy Hypercubes

Efficient communication algorithms [30, 31] have been devised for healthy hypercubes. The standard algorithm for broadcasting a message to all nodes from a single source was presented in [26]. This algorithm embeds a Spanning Binomial Tree (SBT) in the hypercube. The Global Sum (GS) and Global Broadcast (GB) algorithms are the duals of each other.

In the GS operation, data residing in the nodes of a d-cube is collected in a Final Collecting Node (FCN). The GS algorithm requires each node to know the address $(c_{d-1}....c_0)$ of the FCN. For each step i, ($i = d-1, ..., 1, 0$), all nodes with address $(c_{d-1}....c_i\mathrm{x}^{\mathrm{i}})$ receive accumulated

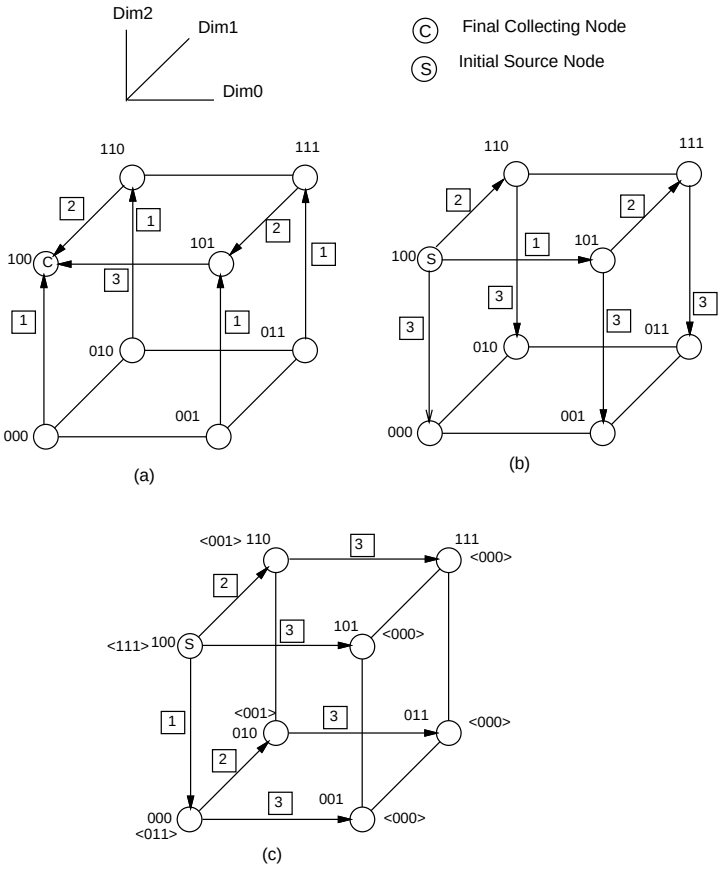

Figure 6: Communication algorithms for a healthy 3-cube (a) GS with dimension sequence $(2, 1, 0)$ (b) GB with dimension sequence $(0, 1, 2)$ (c) GB initiated by ISN

data from nodes with address $(c_{d-1}....\overline{c_i}\mathrm{x}^{\mathrm{i}})$ along dimension i. Hence the data residing in the d-cube is collected in an $i-$subcube at the end of each step. Here, powers of $\{\mathrm{x}, 0, 1\}$ refer to concatenation of $\{\mathrm{x}, 0, 1\}$ and x^0 is the empty string. The communication steps are shown in Fig. 6 (a) for a communication dimension sequence $(2, 1, 0)$ where the FCN is 100. Note that all nodes require the address of the FCN to determine whether to send or receive the data, or to not participate at each step.

In the GB operation, data residing in an Initial Source Node (ISN) is distributed to all nodes in the d-cube. Each node requires to know the address $(s_{d-1}....s_0)$ of the ISN. For each step i, $(i = 0, 1, ..., d-1)$, all nodes with address $(s_{d-1}....s_i\mathrm{x}^{\mathrm{i}})$ send data to nodes with address $(s_{d-1}....\overline{s_i}\mathrm{x}^{\mathrm{i}})$ along dimension i. Data resides in an $(i+1)-$subcube at the end of i steps. The communication steps are shown in Fig. 6 (b) for a communication dimension sequence $(0, 1, 2)$ where the ISN is 100.

The GB operation is initiated by an ISN. If a d-bit control word is sent by a source node to a receiving node indicating how the receiver should continue broadcasting the message, then nodes other than the ISN do not need to know the address of the source. If an intermediate node receives a message with the i^{th} bit of the control word set to 1 (scanning from left), then

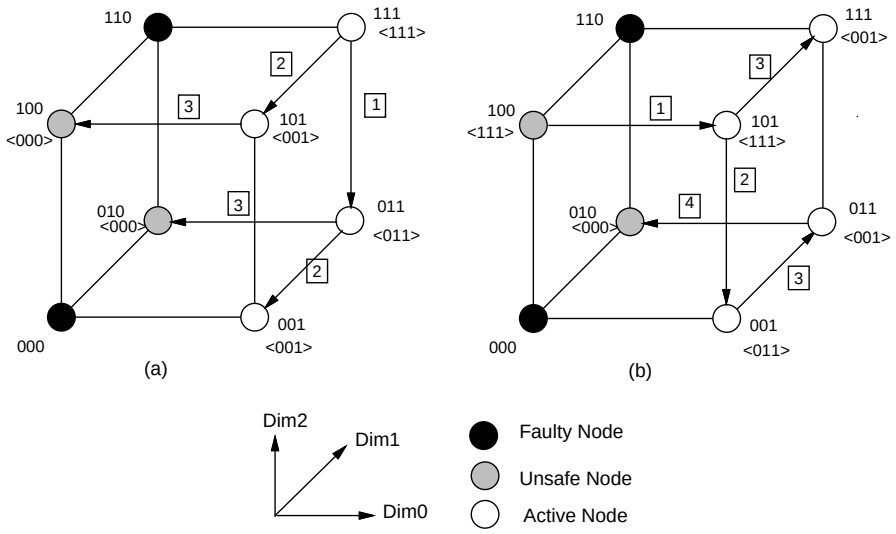

Figure 7: Global broadcast in a faulty hypercube from (a) active ISN (b) unsafe ISN

it sends a copy of the message along dimension i. Each node repeats this step as many times as the number of 1's in the control word. Fig. 6 (c) illustrates this algorithm where the word in the angle brackets ($<>$) is the control word received by the node. The ISN is 100 and the communication dimension sequence is $(2, 1, 0)$.

4.2 Fault-Tolerant Global Broadcast Algorithms

The motivation in fault–tolerant communication algorithm design has been to devise schemes that take the same number of communication steps as the algorithms for healthy hypercubes since an increase in the number of communication steps is a heavy penalty to pay in message passing systems.

Several researchers [32, 27] have investigated GS/GB algorithms for faulty hypercubes. In [32] the GB is performed by constructing a family of *d* link-disjoint (a link is unidirectional and an edge consists of two directed links) spanning trees of height d rooted at the source node. This algorithm tolerates $\lceil \frac{d}{2} \rceil - 1$ edge faults and takes *d* communication steps.

Broadcasting in faulty hypercubes using a d-bit control word was considered in [27], where an *unsafe* node is defined as one that has at least two faulty or unsafe neighbors. A node that is not unsafe is defined as an *active* node. An algorithm that takes $O(d^3)$ communication steps is used to determine the unsafe nodes in a d-cube. The authors show that the GB algorithm takes d communication steps if the source node is not unsafe or faulty (*i.e.*, active) and $d+1$ steps if the source is unsafe provided that the number of node faults does not exceed $\lceil \frac{d}{2} \rceil$. At each step of the broadcast an active node is assigned to broadcast in a subcube, the subcube size indicated by the number of 1's in the control word received by the active node. The control word is scanned from left to right. The algorithm sends a message from an active node to an ***active*** neighbor

along dimension i, if the corresponding bit *i* of the control word is 1; it sends a message to an *unsafe* neighbor along dimension i only if bit *i* of the control word is 1, and all other bits are 0's. In other words, it broadcasts to all unsafe nodes only in the d^{th} step of the GB algorithm. Fig. 7 (a) illustrates a 3-cube with two faults performing the GB from an active ISN 111 in 3 steps. The number in the square denotes the communication step number and the sequence in the angle brackets (<>) is the control word received by the node. The ISN generates a control word of $< 111 >$ and first sends along dimension 3 with the control word altered to $< 011 >$ to reflect the subcubes 1xx and 0xx in which nodes 111 and 011 must continue to broadcast.

If the ISN is an unsafe node, then the ISN sends the message to the first active neighbor obtained by scanning the control word from left to right. This active node acts as a new ISN and performs the GB as before except that no message is sent back to the original unsafe ISN. Fig. 7 (b) illustrates the cube performing the GB from an unsafe ISN *i.e.*, node 100. Note that in step 4 node 101 does not send the message to 100 since 100 is the original ISN.

4.3 Global Sum for Faulty Hypercubes

In Section 4.1 a d-step algorithm for the GS operation in healthy hypercubes was described. The GS [24] can also be collected by performing a partial Global Sum operation in 2^i disjoint $(d-i)$-subcubes in parallel, collecting the partial global sum in nodes that form an i-subcube. For example, in a 4-cube, for $i = 1$, splitting the 4-cube along dimension 3, results in two disjoint 3-cubes 0xxx and 1xxx. Each subcube can perform a partial GS in 3 steps collecting in nodes 0000 and 1000 respectively, which are the nodes of the 1-cube x000. Then, these $2^i = 2$ nodes of the 1-cube x000 perform a GS operation in 1 step. Note that we can split the cube into disjoint subcubes of different sizes and yet apply the same principle. For example, the 3-cube 1xxx obtained earlier can be split along dimension 2 into two 2-cubes 10xx and 11xx. Now, the partial GS can be performed in 2 steps in the three subcubes $S_1 = 0$xxx, $S_2 = 11$xx, and $S_3 = 10$xx in parallel, with nodes $(0000$ and $0100 \in S_1)$, node $(1100 \in S_2)$ and node $(1000 \in S_3)$ collecting the partial results. These are nodes of the 2-cube xx00 and can perform a GS in 2 steps. Note that in general, the dimension sequence of the partial GS can be different for the disjoint cubes. This technique can be used in faulty hypercubes where the d-cube is split into good and faulty partners as explained next.

Consider the 4-cube shown in Fig. 8(a) with 3 faulty nodes at $f_1 = 0010$, $f_2 = 1100$, and $f_3 = 1111$. The idea is to try to split the 4-cube into two 3-cubes, one healthy and the other containing all faulty nodes. If such a split can be found, then the healthy nodes in the faulty cube will send to the adjacent nodes in the good cube in one step and the GS will be performed in the good cube in $d-1$ communication steps. A $(d-1)-$cube containing all of the faulty nodes can be found by applying the intersection operation $\mathbf{I_d} = \{\mathbf{I}\}^{\mathbf{d}}$ defined in Section 3 ($0I0 = 0, 1I1 = 1, 0I1 = q, qI0 = qI1 = qIq = q$). The bound coordinates in the intersection

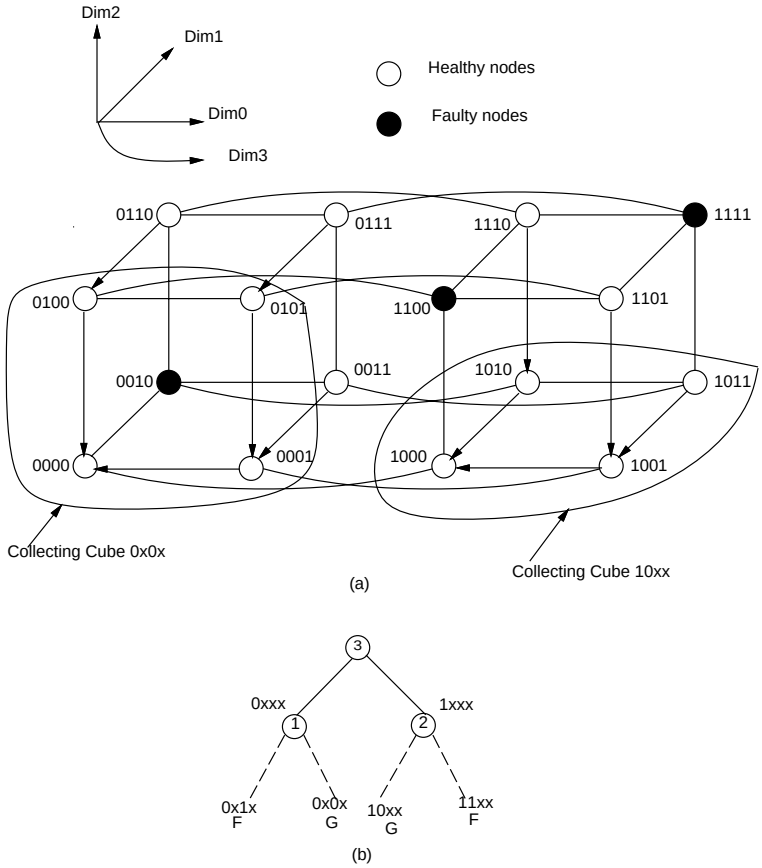

Figure 8: (a) Performing 4 step GS in a faulty 4-cube (b) Split tree for the GS operation shown in (a)

of all faulty processors indicate the fixed coordinates among them, and can be used to split the cube. For the 4-cube considered, $I_{1,2,3} = f_1 I f_2 I f_3 = qqq$, and therefore a good-faulty subcube split cannot be found. Hence the 4-cube is split along any dimension into two faulty 3-cubes. Fig. 8(b) shows the first split along dimension 3 resulting in two 3-cubes $1\mathrm{xxx}$ and $0\mathrm{xxx}$ where f_2 and f_3 are contained in $1\mathrm{xxx}$ and f_1 is contained in $0\mathrm{xxx}$. Since $0\mathrm{xxx}$ contains only one fault, we can choose any dimension, for example dimension 1, to split it into good-faulty partners $0\mathrm{x}0\mathrm{x}$ and $0\mathrm{x}1\mathrm{x}$ respectively. $1\mathrm{xxx}$ has to be split again and dimension 2 which is a bound coordinate of the intersection ($I_{2,3} = 11qq$) is chosen. This results in good-faulty partners $10\mathrm{xx}$ and $11\mathrm{xx}$ respectively. Now, the healthy nodes in the faulty subcubes can send their data to their partner in the good subcubes in parallel in 1 step. The partial GS's for the subcubes $0\mathrm{x}0\mathrm{x}$ and $10\mathrm{xx}$ can be performed in parallel in 2 steps. In order to collect the results of the 2 partial GS's in one node in the 4th step, the nodes collecting the partial GS's in each 2-cube must be adjacent, *i.e.*, the collecting nodes containing the result of the partial GS's must be in the same 1-cube, so that the final GS can be collected in 1 step. Such collecting nodes can be determined by performing a consistency operation on the healthy (collecting) subcubes found in the last split. For the two collecting cubes $10\mathrm{xx}$ and $0\mathrm{x}0\mathrm{x}$ in Fig. 8(a), the collecting nodes can be found by performing a *consistency* operation $\&_d = \{\&\}^d$ defined as follows: 0&0=0&x=0,

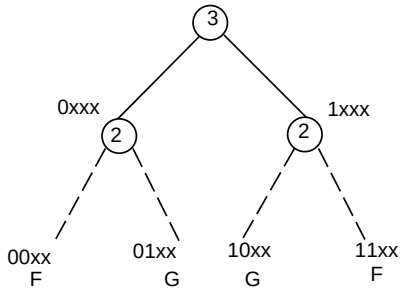

Figure 9: Other possible way to split the faulty 4-cube

1&1=1&x=1, 0&1=–&0=–&1=–&x=–, x&x=x. The candidate nodes for each collecting cube are obtained by replacing the "$-$" coordinates of the result with the corresponding coordinate values in the collecting cube. For this example, $0\mathrm{x}0\mathrm{x}\&10\mathrm{xx} = -00\mathrm{x}$. The candidate collecting nodes $(000\mathrm{x} \in 0\mathrm{x}0\mathrm{x})$ and $(100\mathrm{x} \in 10\mathrm{xx})$ form a 1-cube called the Final Collecting Cube (FCC). Fig. 8(b) shows the splitting operation. The *split tree* is a binary tree where each node represents a split and has associated with it a dimension d_i used at that split to partition the cube. Note that the split tree does not include the edges of the tree corresponding to the last splitting operation to obtain the good-faulty partners (shown in dotted lines) since these edges do not have a node at either end. A *leaf* of the split tree is a node of degree 1, while an *internal node* is a node of degree 2 (root) or 3.

Now consider the same example but with dimension 2 chosen to split both subcubes $0\mathrm{xxx}$ and $1\mathrm{xxx}$ into good-faulty partners. The split tree in Fig. 9 shows that the 4-cube can be split resulting in two good 2-cubes $01\mathrm{xx}$ and $10\mathrm{xx}$, but partial collecting nodes that are adjacent cannot be found.

4.4 d-Step Fault-Tolerant GS/GB Algorithm

By using a *unique* dimension at each split, the GS/GB operation can always be performed in d steps if the number of faults is $\leq \lceil \frac{d}{2} \rceil$. Having split the cube, the data from each healthy node in the faulty subcube is sent to its partner in the good subcube along a dimension *i*, where *i* is the dimension used at the final split to obtain a good-faulty pair. This is done in one step. Now all the data resides in the good subcubes. The following Depth First Search (DFS) operation on the split tree yields the dimension set D_c of the FCC. Initially all the nodes of the tree are uncolored and D_c is empty. When a node is visited, if it is not colored, and it is not a leaf node, then it is colored and the dimension associated with that node is added to set D_c. The dimension of the FCC, which is the number of internal nodes in the split tree is denoted as k where $k = \mid D_c \mid$. Note that the dimension set of the FCC can also be obtained by the & operator applied to the good subcubes. The data in each good subcube is collected in parallel by a partial GS performed in $d - k - 1$ steps. Let the set of free coordinates of the collecting good subcubes be denoted by D_{gi} where *i* is the i^{th} good subcube. The dimension sequence along which to perform the

partial GS operation in the i^{th} good subcube is determined by deleting from D_{gi} the dimensions in the intersection of D_{gi} and D_c. By construction we can show that the collecting node(s) in the good subcubes form a k dimensional Final Collecting Cube (FCC). The GS operation can then, be performed on the FCC in *k* steps to collect the data in one node. Hence, the entire operation takes d steps. The formal proofs can be found in [33].

The following example illustrates the methodology used to determine the FCN and the dimension sequence along which to perform the first communication step from faulty to good subcubes, and thereafter the partial GS in each good subcube.

Fig. 10 shows a 5-cube with 3 faults f_1 at 00100, f_2 at 00010, and f_3 at 11000. Fig. 11 shows the split tree for a chosen split. The FCN that collects the result of the GS operation is determined from the constructed split tree. Each node (circle) represents a subcube and the number inside the circle represents the dimension along which that cube is split. First the cube is split along dimension 4, giving rise to two faulty cubes 1xxxx and 0xxxx. Next, the first faulty subcube (1xxxx) is split along dimension 3 and the other faulty subcube (0xxxx) along dimension 2. The first split gives rise to a good-faulty pair where the good subcube is denoted by $G_1 = 10$xxx. The second faulty subcube is split into two faulty subcubes 0x1xx and 0x0xx. Finally, the faulty subcubes 0x1xx and 0x0xx are split along dimensions 1 and 0 respectively to give good subcubes $G_2 = 0$x11x and $G_3 = 0$x0x1 respectively. Note that this is only one of the $d!$ ways to split the 5-cube into good-faulty partners where a unique dimension is chosen for each split.

Using the DFS algorithm on the split tree, $\{D_c\} = \{4, 2\}$. The dimension of the FCC, is $k = \mid D_c \mid = 2$. The FCC is given by x0x11 where the fixed dimensions correspond to the dimensions that result in the good-faulty splits, and the free dimensions correspond to the internal nodes of the split tree.

Any node contained in the FCC can be chosen as the FCN. Fig. 10 shows the GS operation for the 5-cube with the FCN being 00111. The number *i* in the square denotes the communication step *i* involved in the operation. In the first step, each node in the good subcubes receives data from its healthy partner in the faulty subcubes along the dimension used to perform the final split. Nodes in G_1 receive from their partners along dimension 3, while nodes in G_2 and G_3 receive from their partners along dimensions 1 and 0 respectively. This takes one communication step. The partial GS in each good subcube is performed in $d - k - 1$ steps *i.e.*, 2 in this case.

The dimension sequence D_i along which to perform the partial GS operation in the i^{th} good subcube must satisfy the condition $D_i \cap D_c = \phi$. Hence, in G_1 the GS is performed along dimensions 0 and 1, in G_2 along 0 and 3, and in G_3 along 1 and 3. The action to be taken by a node *i.e.*, send, receive, or not participate is determined as in the healthy GS algorithm using the address(es) of the collecting node(s) in each subcube. The FCC is x0x11. The collecting nodes in G_1 are 10011 and 10111, the collecting node in G_2 is 00111, and the collecting node in

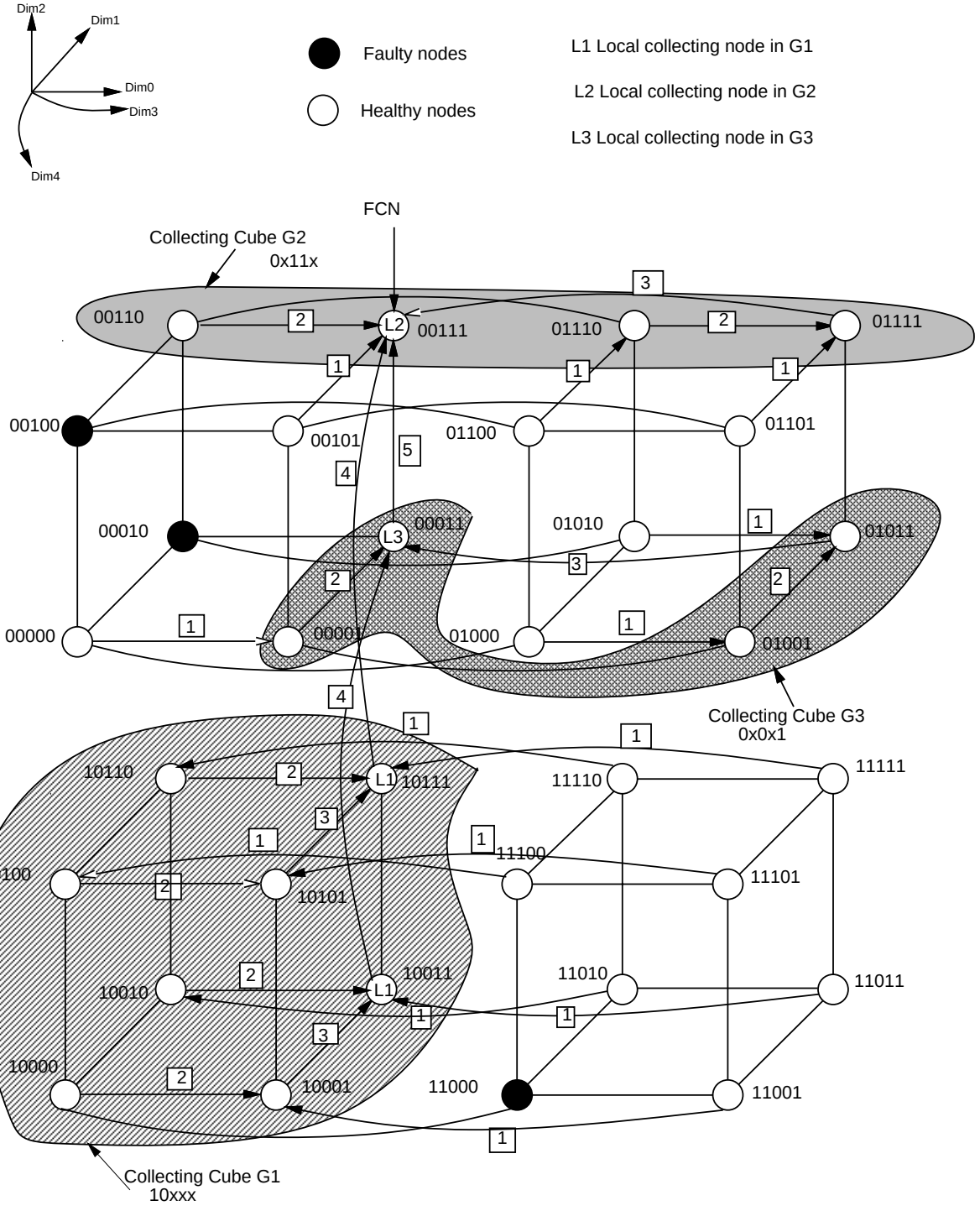

Figure 10: Hypercube of dimension 5 with three faulty nodes performing GS operation

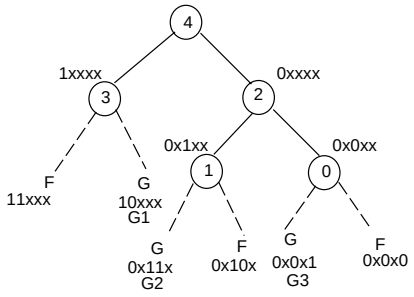

Figure 11: Split tree for the faulty 5-cube in Fig.10

G_3 is 00011. After the partial collection in the collecting nodes, the GS operation is performed on the FCC in 2 steps collecting the data in the FCN. The total number of steps taken is 5 (1 step for sending from faulty to good subcubes $+$ 2 steps to perform the local partial GS in each good subcube $+$ 2 steps to perform the GS in the FCC).

5 Fault-Tolerant Hypercubes

As mentioned in Section 1, faulty nodes can substantially reduce the dimension of the largest fault-free available subcube. To preserve the dimensionality of the faulty hypercube, hardware redundancy and spare allocation schemes [8, 9, 10, 7, 11, 12, 13, 14, 15] have been investigated.

A hardware scheme was first proposed by Rennels [8]. Here, $N = 2^d$ processors are grouped into $S = 2^s$ clusters of $M = 2^m$ processors each, where $d = m + s$ and one spare is assigned per cluster. Crossbar switches are employed to add spare nodes via the additional port in dimension $(d + 1)$ (Fig. 12). If a processor fails in a cluster, the spare replacing it must be able to communicate in d dimensions. Therefore, it has to be linked to the m neighboring processors in that cluster as well as one processor in each of the s external subcubes to which the failed processor is connected. This is accomplished using two crossbars, namely the *CCB (Connection Crossbar)* and the *RCB (Relay Crossbar)*. The *CCB* allows a spare to be connected to each of the 2^m processors in its designated cluster via their spare port. Using the *RCB*, the spare is linked to each of the other s clusters to which the host cluster is connected. This approach can tolerate only a single faulty node per cluster with a significant overhead. It does not tolerate any link failures. For better fault coverage, spare boards consisting of one spare processor and S crossbars are suggested. Each spare node then can replace any faulty node directly. However, the degree of the spare node would effectively be equal to the size of the hypercube. Furthermore, additional crossbars to interconnect the spares are needed.

Several approaches are proposed by Banerjee et. al. [10, 7, 12] to perform reconfiguration using spare nodes and spare links. In [12] two spares per 3-cube are assigned (Fig. 13) and the spare processors form a $(d-2)$-cube. Upon a node failure, the faulty processor is replaced with the local spare. The link connecting the spare to the faulty processor and the link connecting

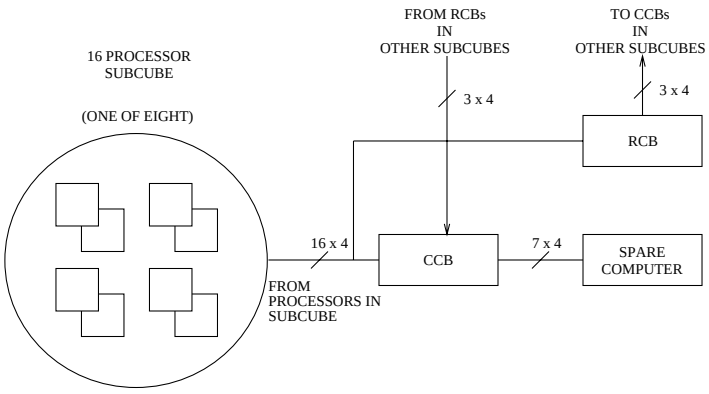

Figure 12: Rennels' scheme for assigning a spare to a cluster

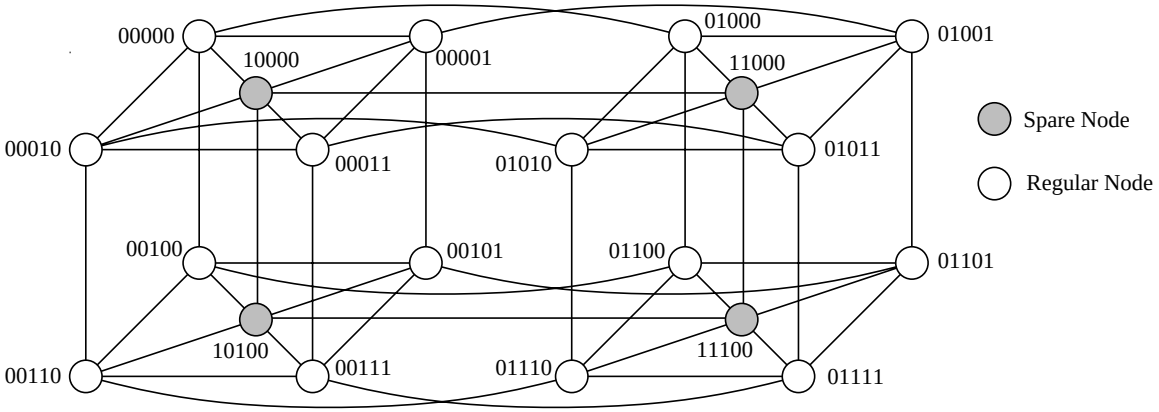

Figure 13: Augmented four-dimensional hypercube topology proposed by Banerjee

it to the processor diagonally opposite to the faulty processor are disabled. All other links are enabled. The spare node (S) which replaces the faulty one sends its address and the address of the faulty node to all the spare nodes connected to S. The function of each of the other spares is now that of a simple switch. Consequently each spare determines the node to which it needs to be connected by using the bitwise *XOR* of its own address, the address of the spare S, and the address of the faulty node. As an example let us consider Fig. 13 when node 01100 is faulty. The reconfiguration steps are as follows. Replace processor 01100 with the spare processor 11100. The links between the spare 11100 and nodes 01100 and 01111 are disabled. All other links are enabled. The spare node 11100 sends its address as well as the address of the faulty node (01100) to the its neighboring spare nodes (11000 and 10100). The spare nodes 11000 and 10100 calculate the address of the nodes they should be connected to as: $01100 \oplus 11100 \oplus 11000 = 01000$ and $01100 \oplus 11100 \oplus 10100 = 00100$. The spare nodes 11000 and 10100 then act as switches. This system can only tolerate a single node failure.

In [10] Banerjee proposes two schemes to tolerate faulty processors. The first scheme uses two sets of nodes; *P-nodes* and *S-nodes*. *P-nodes* are the regular nodes of the hypercube. An *S-node* consists of a spare attached to a *P-node*. *S-nodes* are allocated such that every regular node is at a Hamming distance of one from a spare node. For example in a 3-cube, spares can

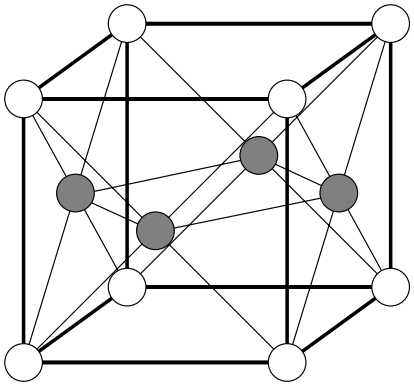

Figure 14: A 3-dimensional reconfigurable hypercube proposed by Alam and Melhem

be attached to nodes 0 and 7. When a node fails, the spare node at a distance of one from the faulty node is brought on line. The *S-node's* communication module is used to handle both the communication needs of the faulty node and that of the attached regular node. Messages that would have been routed to the faulty node, need to be routed to the spare using a routing table [7]. Therefore, some physical links may experience added traffic which is defined as *congestion*. If more than one node fails and the failed nodes are at a Hamming distance of one from an *S-node*, a weighted bipartite graph is set up to match the faulty nodes with the spares with the minimum *link dilation*. The link dilation in a faulty hypercube is defined by $d(v, \phi(v))$ where v represents the faulty node, $\phi(v)$ identifies the spare replacing v, and d denotes the distance between them. Both dilation and congestion are high with this scheme.

In the second method, spare nodes are placed between links connecting pairs of nodes. For example in a *3-cube*, one spare can be placed between nodes 0 and 1, and another spare between nodes 6 and 7. Similar to the previous scheme, a weighted bipartite graph is used to assign spares to the faulty nodes. Again the scheme results in high dilation and congestion.

Alam and Melhem [34, 13, 14] using hardware redundancy, developed augmented approaches to tolerate faulty nodes. In [34, 13], they have proposed two schemes. In the first one, a single spare is added to each cluster of 4 nodes. If one of the four nodes fails, the spare replaces it and inherits its address. The *e-cube* routing algorithm is no longer valid. The new routing algorithm takes up to $2d + 2$ steps as compared to the d steps of the *e-cube* algorithm, to send a message from a source to a destination. The system can tolerate one faulty node per cluster. To allow more faulty nodes per cluster, in the second scheme 50% redundancy is used as shown for $d = 3$ in Fig. 14. Note that each node can be replaced by one of two spares, making the spare assignment nondeterministic. Therefore, an even more complicated routing algorithm is needed. The system can handle two faulty nodes per cluster.

In [14] two spares are assigned per cluster of 4 nodes. Each spare is connected to every node in its cluster using multiple links (channels). The spares are also connected to form a cube of their own. Upon a node failure, the faulty node is assigned to one of the spares within the

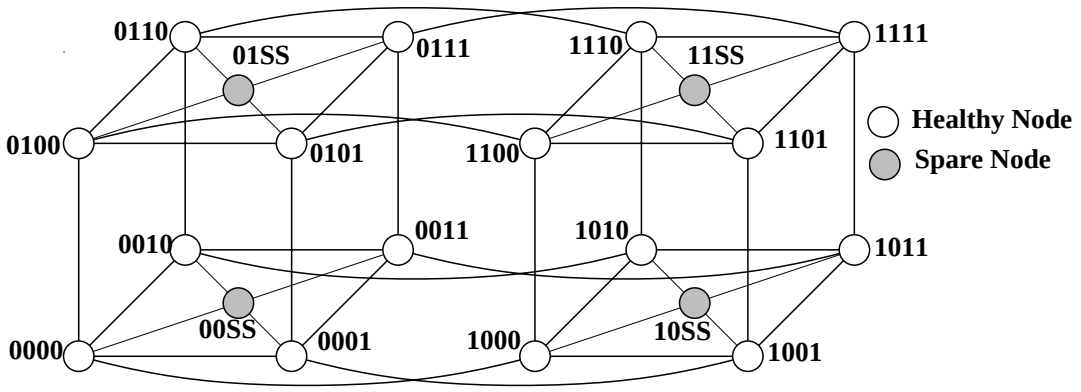

Figure 15: Hypercube of dimension 4 with clusters of dimension 2

cluster. The links of the faulty node are then discarded and the mirror image of the spare in the other clusters are used to connect the spare to the neighboring nodes of the faulty node. Since more than one neighbor of a node could be faulty, multiple channels are necessary to connect the node to the spares of the faulty nodes. Three faults within a cluster is fatal.

In [15] two augmented schemes are proposed to tolerate faulty nodes and/or links. The presented schemes tolerate a large number of faults without any performance degradation and the resultant configuration does not affect either the communication or computational algorithms already developed for the hypercube multiprocessor. In the first scheme a single spare is assigned per cluster of nodes (Fig. 15). The allocation of spares is facilitated by using a routing element on each node similar in concept to the *Direct Connect Module (DCM)* [25] of *Intel* hypercubes. It is assumed that the faulty nodes retain their communication capability. A spare can logically replace the faulty node within its designated cluster. For example, in Fig. 15, the spare $01SS$ may logically replace any one of nodes $0100, 0110, 0111, 0101$. Upon detecting a node failure, the spare activates its link to the faulty node and disables the rest of its links. The spare's communication module then sends/receives its data to/from other nodes via the *DCM* of the faulty node.

One intra-cluster link failure per cluster can be tolerated. Upon detection of a link failure, the router of the spare node is used to establish a parallel path to the faulty link. The processing elements at the two ends of the faulty link then logically replace their *channel routing elements* [15], which connect them to the faulty link with the *spare channel routing element*. An inter-cluster link failure is fatal.

In the second scheme, the spares are connected to form a $(d-2)$-cube. The approach can tolerate a larger number of faulty nodes [**?**], by establishing dedicated paths, in the spare hypercube, between the spare of the faulty cluster and spares of the non-faulty clusters. The approach can also tolerate both intra-cluster and inter-cluster link failures by establishing parallel path(s) to the faulty link(s). Fig. 16 demonstrates the reconfiguration of a hypercube upon detection of faults in links $0-00$ (the link between the nodes 0100 and 0000), $100-$ and nodes 1011, 1100,

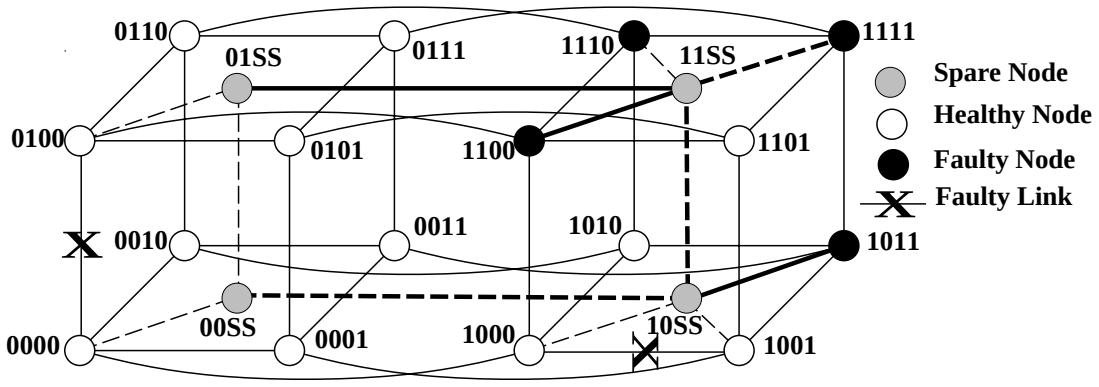

Figure 16: Reconfiguration of faulty hypercube

1110. Multi-channel communication capability of the spare node $10SS$ is used to logically replace the node 1011 and at the same time establish a path between the nodes 1001 and 1000. As shown in Fig. 16 the faulty inter-cluster link $0 - 00$ is replaced by the path connecting the nodes $0100,\ 01SS,\ 00SS,$ and 0000. Similarly, spare nodes $01SS$ and $11SS$ logically replace faulty nodes 1100 and 1110, shown by the dark and dashed lines respectively.

6 Conclusions

A faulty hypercube needs to be reconfigured to perform the computational task and communication as required by the algorithm, with minimal or no performance degradation. The first approach is to avoid the faulty nodes/links and perform useful computation with the fault-free nodes. This paper outlines research in the area of embedding topologies such as lower dimensional hypercubes, rings, meshes, and trees in the faulty hypercube. To support the communication primitives required by the algorithm, a scheme that performs the Global Sum/Global Broadcast operation in a faulty d-cube is proposed. The second approach is to use hardware redundancy in the form of spare nodes and links that logically replace the faulty node(s)/link(s). A survey of various schemes is presented.

References

[1] E. Dilger and E. Amman, "System level self diagnosis in n-cube connected multiprocessor networks," in *Proc. 14th Int. Symp. on Fault Tolerant Computing*, pp. 184–189, 1984.

[2] C. Aykanat and F. Özgüner, "A concurrent error detecting conjugate gradient algorithm on a hypercube multiprocessor," in *IEEE 17th International Symposium on Fault Tolerant Computing*, pp. 204–209, July 1987.

[3] C. Aykanat, F. Özgüner, P. Sadayappan, and F. Ercal, "Iterative algorithms for solution of large sparse systems of linear equations on hypercubes," *IEEE Transactions on Computers*, vol. c–37, pp. 1554–1568, December 1988.

[4] F. Özgüner and C. Aykanat, "A reconfiguration algorithm for fault tolerance in a hypercube multiprocessor," *Information Processing Letters*, vol. 29, pp. 247–254, November 1988.

[5] B. Becker and H. U. Simon, "How robust is the n-cube?," *Proc. 27th Annu. Symp. Foundations Comput. Sci.*, pp. 283–291, October 1986.

[6] C. C. Li and W. K. Fuchs, "Graceful degradation on hypercube multiprocessors using data redistribution," *Proceedings of the Fifth Conference on Hypercube Concurrent Computers and Applications*, pp. 1446–1454, April 1990.

[7] P. Banerjee, "Reconfiguring a hypercube multiprocessor in the presence of faults," *Proceedings of the Fifth Conference on Hypercube Concurrent Computers and Applications*, pp. 95–102, 1990.

[8] D. Rennels, "On implementing fault-tolerance binary hypercubes," *Proceedings of the IEEE International Symposium on Fault Tolerant Computing*, pp. 344–349, 1986.

[9] S. C. Chau and A. L. Liestman, "A proposal for a fault-tolerant binary hypercubes architecture," *Proceedings of the IEEE International Symposium on Fault Tolerant Computing*, pp. 323–330, 1989.

[10] P. Banerjee, "Strategies for reconfiguring hypercubes under faults," *Proceedings of the IEEE International Symposium on Fault Tolerant Computing*, pp. 210–217, 1990.

[11] A. Witkowski and R. Lee, "Fault tolerance for the hypercube multiprocessor," *Proceedings of the Fifth Conference on Hypercube Concurrent Computers and Applications*, pp. 117–122, 1990.

[12] P. Banerjee, J. Rahmeh, C. Stunkel, V. Nair, K. Roy, V. Balasubramanian, and J. Abraham, "Algorithm-based fault tolerance on a hypercube multiprocessor," *IEEE Transactions on Computers*, vol. 39, pp. 1132–1145, September 1990.

[13] M. Alam and R. Melhem, "An efficient modular spare allocation scheme and its application to fault tolerant binary hypercubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 117–126, January 1991.

[14] M. Alam and R. Melhem, "Channel multiplexing in modular fault tolerant multiprocessors," *Proceedings of the IEEE International Conference on Parallel Processing*, pp. I492–I496, 1991.

[15] B. Izadi and F. Özgüner, "Spare allocation and reconfiguration in a fault tolerant hypercube with direct connect capability," *Proceedings of the Sixth Conference on Distributed Memory Computing Conference*, pp. 711–714, April 1991.

[16] F. Harary, J. P. Hayes, and H. J. Wu, "A survey of the theory of hypercube graphs," *Computers and Mathematics with Applications*, vol. 15, pp. 277–289, 1988.

[17] A. Wu, "Embedding of tree networks into hypercubes," *Journal of Parallel and Distributed Computing*, vol. 2, pp. 238–249, April 1985.

[18] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Transactions on Computers*, vol. c–37, pp. 867–872, July 1988.

[19] S. K. Chen, C. T. Liang, and W. T. Tsai, "An efficient multi-dimensional grids reconfiguration algorithm on hypercubes," *Proc. of 18th Fault Tolerant Computing*, pp. 368–373, June 1988.

[20] T. C. Lee, "Quick recovery of embedded structures in hypercube computers," *Proceedings of the Fifth Conference on Hypercube Concurrent Computers and Applications*, pp. 1426–1435, April 1990.

[21] S. R. Deshpande and R. M. Jenevein, "Scalability of a binary tree on a hypercube," *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 661–668, 1986.

[22] F. J. Provost and R. Melhem, "A distributed algorithm for embedding trees in hypercubes with modifications for run-time fault tolerance," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 85–89, February 1992.

[23] M. Y. Chan and S. J. Lee, "Distributed fault-tolerant embeddings of rings in hypercubes," *Proceedings of the Fifth Conference on Hypercube Concurrent Computers and Applications*, pp. 834–838, April 1990.

[24] J. Wang and F. Özgüner, "Embeddings, communication and performance of algorithms in faulty hypercubes," *Proceedings of the Fifth Conference on Hypercube Concurrent Computers and Applications*, pp. 1455–1464, 1990.

[25] S. Nugent, "The *iPSC/2* direct-connect communication technology," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 51–60, January 1988.

[26] H. Sullivan, T. Bashkow, and D. Klappholz, "A large scale, homogeneous, fully distributed parallel machine," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 105–124, March 1977.

[27] T. C. Lee and J. P. Hayes, "Routing and broadcasting in faulty hypercube computers," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 346–354, January 1988.

[28] M. Y. Chan and S. J. Lee, "Fault-tolerant embeddings of complete binary trees and rings in hypercubes," *Technical Report UTDCS-17-89 University of Texas at Dallas*, August 1989.

[29] F. S. Roberts, *Applied Combinatorics*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[30] Y. Saad and M. H. Schulz, "Data communication in hypercubes," *Tech. Report YALEU/DCS/RR-389, Dept. of Computer Science*, June 1985.

[31] S. L. Johnsson and C. T. Ho, "Optimum broadcasting and personalized communication in hypercubes," *IEEE Transactions on Computers*, vol. C-39, pp. 1249–1268, September 1989.

[32] J. Bruck, "Optimal broadcasting in faulty hypercubes via edge-disjoint embeddings," *Tech. Report RJ 7174(67394), IBM, Computer Science*, November 1989.

[33] S. Balakrishnan and F. Özgüner, "An n-step global sum/global broadcast algorithm for n-dimensional faulty hypercubes," *Tech. Report Dept. of Electrical Engineering The Ohio State University*, January 1992.

[34] M. Alam and R. Melhem, "Fault tolerance and reliable routing in augmented hypercube architectures," *IEEE 18th Annual Phoenix Int. Conf. on Computer Communication Proceeding*, pp. 19–23, 1989.