23 Jun 1993

A General Method for Maximizing the Error-Detecting Ability of Distributd Algorithms

Martina Schollmeyer

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

A GENERAL METHOD FOR MAXIMIZING THE ERROR-DETECTING ABILITY OF DISTRIBUTD ALGORITHMS†

Martina Schollmeyer and Bruce McMillin

CSC-93-16

June 23, 1993

Department of Computer Science
University of Missouri at Rolla
Rolla, Missouri 65401

† This work was supported in part by the National Science Foundation under Grant Numbers MSS-9216479 and CDA-9222827, and, in part, from the Air Force Office of Scientific Research under contract numbers F49620-92-J-0546 and F49620-93-I-0409.

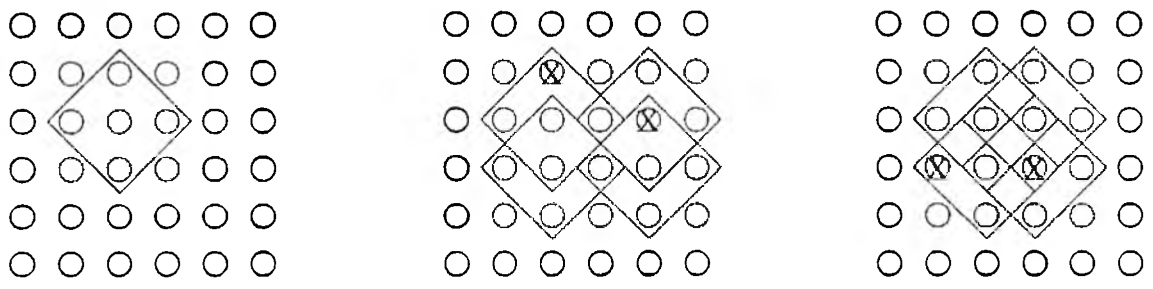
Abstract

Error-detecting algorithms can determine when, at run time, a program deviates from its expected behavior due to a hardware, software or communication error. In a fixed interconnect multiprocessor system, the error detecting ability heavily depends on the number of faults, which is bounded, and their spatial distribution. Otherwise multiple fault occurrences can mask each other. This paper provides a general method for computing the overall system failure bound, the maximal fault index, from the system topology and local communication patterns. The result of the computation is used to design a mapping of processes to processor groups such that multiple processor failures preserve the error-detecting ability of the algorithm. We show the problem of finding the maximal fault index to be NP-Hard and show, for certain regular topologies, that the problem yields polynomially computable embeddings. Finally, we give an example of mapping an error detecting matrix relaxation algorithm derived from program verification using Changeling.

I. Introduction

Error-detecting algorithms work by checking assertions, at run time, to detect hardware, communication [JoAb87], and software errors [McNi88]. A properly chosen set of assertions, such as those generated from program verification, guarantees that, when operationally evaluated, as in Changeling [LuSM92], the program meets its specifications. However, there exist bounds on the spatial locality of faults which, if exceeded in a fixed topology, nullify the error detecting ability of the algorithm. This paper provides a formal assessment technique on the error-detecting abilities of an algorithm which can be used in mapping the algorithm to the computational processors such that its error-detecting abilities are preserved.

A communication environment of a specific processor is the set of processors induced by the local interprocess communication dependencies of the algorithm (Figure 1.1.a shows the *star pattern*). Given, as a design parameter, the maximum number of faults that can be permitted in each communication environment such that all errors can still be detected, the *local fault measure* t_g, and the topology of the entire system, we can compute the *global fault measure* t_s which maximizes the number of permitted faults in the system while maintaining the local fault measure condition.



a) A communication environment (Star Pattern)

b) Overlapping communication environments with non-interfering faults

c) Overlapping communication environments with interfering faults

Figure 1.1: Communication environments.

Figure 1.1.b shows a scenario where two faulty components in the system will not violate the local fault measure condition. Continuing this analysis over the entire multiprocessor system, for all possible fault patterns, yields t_s. By contrast, Figure 1.1.c shows a syndrome of faults which violates the local fault measure for at least one case.

An optimal fault distribution yields a partitioning of processes into groups. Every process within a particular group can be simultaneously faulty without affecting the error-detecting capability of the algorithm. These groups are mapped disjointly onto the actual processor topology. An example for this is given in Figure 1.2. The communication environment used (*the square*) is described in Figure 1.2.a, the conventional process-to-processor mapping is shown in 1.2.b, and a mapping where processes that may be simultaneously faulty are mapped into the same processor group can be seen in 1.2.c. Details of the mapping algorithm are given in Section IV.

In Section II, we provide definitions for different collections of processors based on their faulty or non-faulty status. Section III gives a graph coloring algorithm for determining the distribution of faulty processors within the topology and Section IV shows that the characterization of an optimal fault distribution is NP-complete and that of finding the maximal fault index is NP-hard for arbitrary topologies and communication patterns. Section IV also gives an algorithm for determining a process to processor group

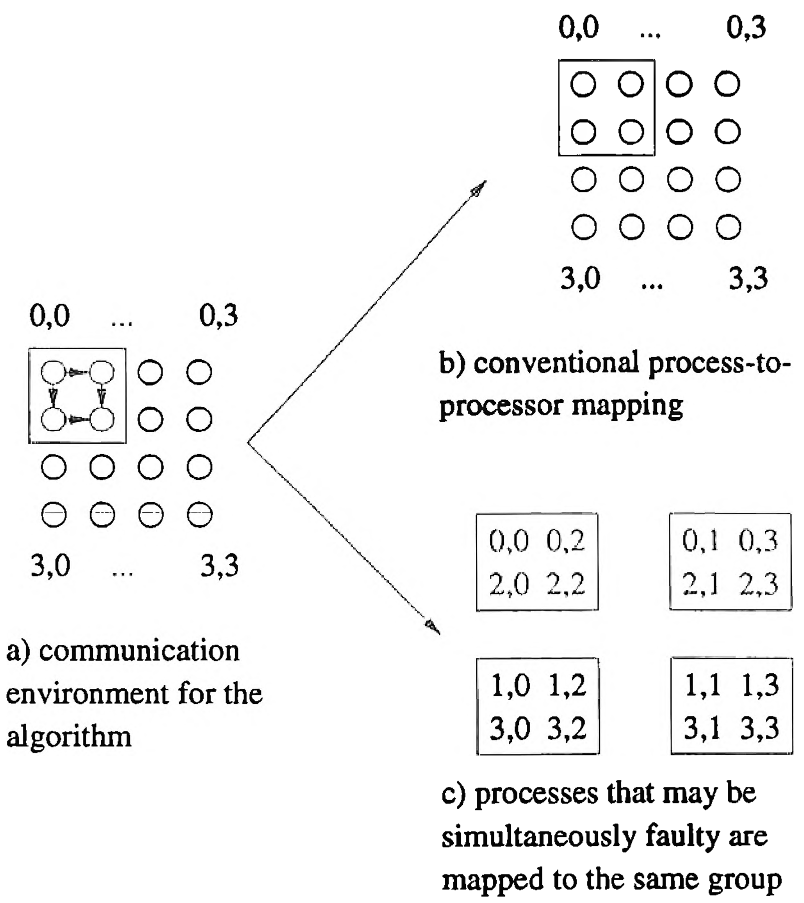

0,0 ... 0,3
3,0 ... 3,3
a) communication environment for the algorithm
0,0 ... 0,3
3,0 ... 3,3
b) conventional process-to-processor mapping
0,0 0,2
2,0 2,2
0,1 0,3
2,1 2,3
1,0 1,2
3,0 3,2
1,1 1,3
3,1 3,3
c) processes that may be simultaneously faulty are mapped to the same group

Figure 1.2: Logical adjacency in the algorithm and physical mapping.

partitioning based on the optimal fault distribution. In Section V we show that the maximal fault index for several specific communication patterns and regular topologies can be found in polynomial time, and we also give partitionings based on their optimal fault distributions. Section VI provides an example of how this form of assessment can be used in an error-detecting matrix relaxation algorithm.

II. Terminology for MPS Topologies

In this paper we use fixed-topology multiprocessor systems as discussed in [FoRa85, Haye76, LeLe85, Rose83]. In contrast to [GuRR93] we do not examine whether an algorithm can detect all combinations of up to k faults where k is a specified bound, but we assume that the algorithm has been designed with a certain local fault tolerance, t_g, for each communication environment [McMi88]. The analysis in [GuRR93] can determine whether every combination of up to t_g faults can be detected, and it provides the minimum number of simultaneous faults for which this condition does not hold any more. Instead, we want to determine the maximum number of faults, t_s and their distribution in the topology for which all errors can still be detected. However, we do not claim that all combinations of up to t_s faults can be tolerated.

The underlying topology of a multiprocessor system (MPS) is described by a graph G(V,E), where the set of vertices V represents the processors in the network and the set of edges E determines the direct communication links between pairs of processors. The network topology of an MPS does not have to be regular, such as a hypercube or mesh, but can be an arbitrary connected graph.

For simplification, we will focus only on processor failures, since a processor failure can be described by the failure of all its links, and a link failure can be described by indicating a processor failure [McNi92]. We will also assume that messages can always be forwarded reliably through intermediate processors, using techniques such as wormhole routing or circuit switching, where messages are passed through designated routers at each processor.

In an MPS interconnection network, the interactions between processors are described by communication patterns. Frequently, algorithms restrict interprocessor communication to adjacent processors to improve efficiency. However, new routing technologies, such as wormhole routing, make the delivery of messages to processors that are a distance of more than one away almost as efficient as direct communication [DaSe86]. We allow for both types of interactions in the communication environment.

Definition: The *communication environment* (CE) of a processor P_i is the set of processors from which P_i will request information during the execution of a program, including P_i. The communication environment of a specific processor is a subset of the set of all n processors in the network, i.e., $CE(P_i) \subseteq \{P_1, P_2, \cdots, P_n\}$. [1]

Definition: A *fault group* of a processor P_i of fault measure t_g, denoted by $FG(P_i)$, is the collection of faulty processors in $CE(P_i)$. To guarantee error detection for all errors

1: Communication environments usually intersect since P_i requests data from other processors and other processors request data from P_i. We need to relate independent failures in different CEs such that the local fault measure, t_g, in each environment is not violated.

caused by these faults, we require that

$$(\forall i,\ 1 \le i \le n)(FG(P_i) \subseteq CE(P_i) \wedge |FG(P_i)| \le t_g\ ).$$

Definition: A collection of processors that must be non-faulty to guarantee detection of all errors induced by the set of faulty processors **P** is called the ***non-fault group*** of **P**, denoted by $NFG(\mathbf{P})$. It is the set that contains all elements in the CEs in which the elements of **P** are members and in which t_g has been reached.

$$NFG(\mathbf{P}) = \bigcup P_j \text{ where } \left(P_j \in CE(P_k) \wedge P_j \notin \mathbf{P} \wedge P_i \in CE(P_k) \wedge P_i \in \mathbf{P} \wedge |FG(P_k)| = t_g \right)$$

For the algorithm to detect all errors, the following must invariantly hold

$$(\forall j)(\mathbf{P} \text{ are faulty} \wedge P_j \in NFG(\mathbf{P}) \rightarrow |FG(P_j)| \le t_g)$$

Depending on the value of t_g, many different non-fault groups exist. The NFG for a set of faulty processors **P** determines on which processors P_j, outside NFG(**P**), the failure of **P** will have no effect. Failures of these components can be tolerated. For an error-detecting algorithm we need to ensure that there will be no conflicts between the faulty processors and their respective NFGs. This means that if a processor fails, it must not be in the NFG of any other failed processor so that detection of all errors induced by the set of faulty processors can be guaranteed.

III. Coloring Faulty MPS Topologies

In this section we discuss how we can find and evaluate the non-fault groups in an interconnection network, based on the individual communication environments.

An *augmentation* of the problem graph represented in the MPS interconnection network adds additional symbolic edges (no augmentation is made to the actual topology) so that the elements located in each CE are adjacent to each other in the augmented problem graph. Thus, each CE forms a completely connected subgraph. The augmented edges correspond to fault dependencies between processors in a CE. Since, at any time, there must be no more than t_g faulty components in each CE, there can be at most t_g faulty vertices adjacent to each non-faulty vertex in the augmented graph, and at most $t_g - 1$ faulty components adjacent to a faulty component. For example, if CE(1)={1,2,3,4} and

CE(7)={2,6,7} then the sets CE(1) and CE(7) will form completely connected subgraphs after the augmentation, and since 2 is a member in CE(1) and also a member in CE(7), it will be adjacent to all processors in CE(1) and CE(7).

Algorithmically, to determine the NFG of an individual processor P_i, we can mark P_i faulty and determine all adjacent nodes in the augmented graph and permit at most $t_g - 1$ of them to be faulty. For $t_g = 1$ and P_i faulty, all other elements in $CE(P_i)$ must be non-faulty, together with all processors P_k where $P_i \in CE(P_j) \wedge P_k \in CE(P_j) \wedge i \neq k$, i.e., all processors that are in a CE with P_i. With $t_g > 1$ there will be many different possibilities to place up to t_g faulty components into each CE.

We use a coloring algorithm to color the graph, indicating faultiness or non-faultiness of components when determining the NFG of an individually faulty processor. We first describe how the coloring will be done for one fault in each CE, i.e. $t_g = 1$, and then extend the algorithm for $t_g > 1$ to multi-coloring, where each vertex has a chromaticity of t_g, to obtain the NFGs. Finally, this algorithm can be used to obtain a possible distribution of component failures for the whole MPS.

The algorithm given in Figure 3.1 describes how to find the NFG for $t_g = 1$ using a coloring algorithm which colors the faulty components in one color and the components that must be non-faulty in a different color. This coloring scheme works for arbitrary communication patterns as long as the CEs of all processors are known.

```
for i:=1 to $n$  /* n is the total # of processors */
    color $P_i$ faulty;
    color all processors which are in a CE with $P_i$ as non-faulty;
    save $NFG(P_i)$; reset colors;
```

Figure 3.1: An algorithm to determine the NFGs for individually faulty processors ($t_g = 1$).

Theorem 3.1: The time complexity of the coloring algorithm is $O(n^3)$.

Proof: Step 2 in the algorithm evaluates at most $(n - 1)$ processors and their CEs, taking $O(n^2)$ steps; the process will be performed a total of n times in the loop. Hence we have a time complexity of $O(n^3)$. □

To extend the algorithm to obtain the NFGs for a larger number of faults per CE, we perform a multi-coloring where each vertex has a chromaticity of t_g. The coloring for a processor P_j is stored in the array color(j, $1..t_g$). If at least one of the colors indicates faultiness then P_j is considered faulty. If all colors show "non-faulty" then P_j must be non-faulty. In any other case we have a "don't care" state since there still exist possibilities to change the fault status of the component. The multi-color algorithm is given in Figure 3.2.

```
for i:=1 to |P|  /* examine the set of faulty processors P */
   /* $P_j$ is the ith element in P */
   color $P_j$ as faulty in color(j,i);
   /* all processors which are in a CE with $P_j$ are adjacent in the augmented graph */
   ($\forall$ $P_k$ adjacent to $P_j$ in augmented graph)(color $P_k$ as non-faulty in color(k,i));

/* determine the fault status of each processor */
for j:=1 to $n$
   $P_j$ := non-faulty;
   for i:=1 to $t_g$
      if color(j,i) = faulty then $P_j$ := faulty; exit i-loop;
      if color(j,i) = don't care then $P_j$ := don't care;
save(NFG(P));
```

Figure 3.2: An algorithm to determine the NFG of a set **P** of faulty processors and arbitrary t_g.

Theorem 3.2: The time complexity of the multi-coloring algorithm for finding one NFG for an arbitrary **P** is $O(n^2)$.

Proof: The loop in the first part of the algorithm examines at most n processors in **P**. Coloring all adjacent vertices in the augmented graph takes at most n steps, giving $O(n^2)$ as complexity for the first part of the algorithm, not considering the time it takes to set up the augmented graph. The second part takes at most $n \cdot t_g$ steps for determining the fault status. Thus, the overall complexity of the algorithm is $O(n^2 + n \cdot t_g) = O(n^2)$, not considering the time to generate the augmented graph. □

To determine a permissible fault distribution for the entire network, we can use the first part of the algorithm given in Figure 3.2; we select an arbitrary processor to become faulty, and keep labeling the NFGs, selecting new faulty components, until there are no undefined color(j,i), $1 \leq i \leq t_g$, labels for any processor P_j. The fault distribution is obtained by determining faulty and non-faulty processors according to the second part of the algorithm in 3.2.

Figure 3.3 shows an example for a 2-coloring, i.e., $t_g = 2$. The dashed lines show the augmentation of each CE. The other edges are the actual links in the network and are not of importance at this stage. The CEs for the processors are as follows: CE(1)={1,2,3}, CE(2)={1,2}, CE(3)={1,2,3}, CE(4)={3,4,5}, CE(5)={3,5,7}, CE(6)={3,4,5,6}, and CE(7)={3,5,7}. Selecting 1 to be faulty in the first pass will cause 2 and 3 to be labeled non-faulty since they are adjacent to 1 in the augmented graph (dashed lines). Then arbitrarily node 5 is chosen to be faulty, forcing 4, 6 and 7 to become non-faulty. This is important to note because although 5 is adjacent to 1, which is faulty, in the original graph, it is not adjacent in the augmented graph. Now all color(j,1) labels for all processors *j* have been filled. In the second pass an arbitrary node is considered faulty, this time 3 is selected. Because all vertices are adjacent to 3 in the augmented graph all of them must be colored non-faulty in color(j,2). This provides a total of three faulty processors, 1, 3 and 5 with at most 2 faulty components in each CE. Components 2, 4, 6 and 7 most be non-faulty.

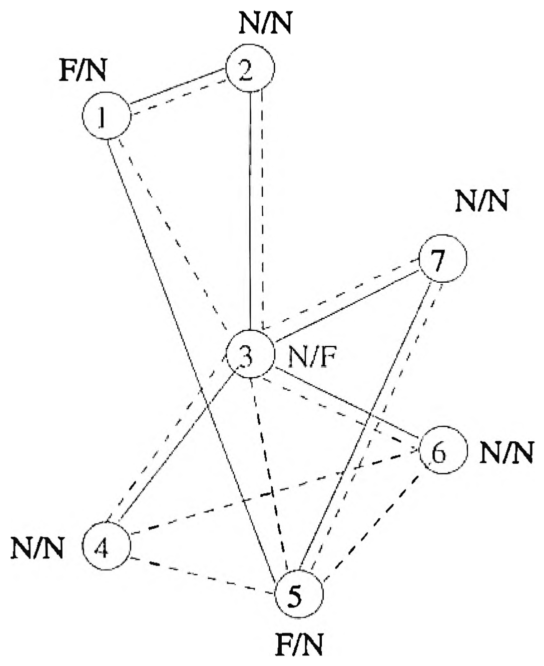


Figure 3.3: A multi-coloring for $t_g = 2$ in an augmented graph.

Theorem 3.3: The time complexity for finding a possible fault distribution using the multi-coloring algorithm is $O(t_g \cdot n^3)$.

Proof: From Theorem 3.1 we can see that it takes $O(n^2)$ steps to find the NFG of one faulty processor. When determining a fault distribution for the whole topology, the vertices are colored until all variables color(j,i) have values assigned to them. In the first round, one node is arbitrarily selected to be faulty and its NFG is colored. Next we color one of the unmarked processors as faulty, find its NFG and color it correspondingly. This process is repeated until all variables have values assigned to them. This takes $O(n^3)$ steps to fill one set of variables color(j,i), where $1 \leq i \leq t_g$. The coloring process is performed t_g times until all color(j,1..t_g) are colored. The determination of the fault status of each processor is done according to the second part of the multi-coloring algorithm with complexity $O(n \cdot t_g)$. Thus, the complexity of finding a possible fault distribution is $O(t_g \cdot n^3)$. □

We now present the NFGs of the processors in a structured way which is useful for determining the maximal fault index of an MPS. There are only three different processor states for each processor with respect to a specific NFG: faulty, non-faulty, or don't care, and will therefore use a matrix representation.

Definition: A *fault matrix* of an MPS gives, for all sets of faulty processors **P**, all processors that must be non-faulty (indicated by the logical value **F** in the matrix) if the elements in **P** are faulty. The faulty processors are marked by **T**, the processors outside the non-fault group are marked by "-". A fault matrix corresponds to a collection of NFGs for a specific t_g.

For $t_g = 1$ there exists only one NFG per processor. For $t_g > 1$ several different NFGs may be found since up to t_g processors can be faulty in each CE. The representation in Figure 3.4 shows the setup of the fault matrix for a 5x5 torus-connected mesh, where all adjacent processors communicate, i.e., the star pattern introduced in Section I. The mesh is labeled row by row from left to right, starting with node 1 at the top left corner, ending at node 25 at the bottom right.

```
              ┌ TFFFFFF--FF----F----FF--F ┐
              │ FTFFFFFF---F----F---FFF-- │
FM(t_g = 1) = │ FFTFF-FFF---F----F---FFF- │
              │           ....            │
              └ F--FF----F----FF-FFFFFFFT ┘

              ┌ ┌ TTF-FFF-------------FF--- ┐ ┐
              │ │ TFT---F--------------F--- │ │
              │ │ T--TF----F--------------F │ │
              │ │           ....            │ │
              │ └ T--FF----F-----F----FF--T ┘ │
              │ ┌ FTTF--FF-------------FF-- ┐ │
              │ │ -TFT---F--------------F-- │ │
              │ │ FT--TF--------------F---- │ │
              │ │           ....            │ │
FM(t_g = 2) = │ └ -T--F---------------FF--T ┘ │
              │                               │
              │ [ -FTTF--FF-------------FF- ] │
              │ [           ....            ] │
              │                               │
              │             ....              │
              │ [ F--FF-------------FF--FTT ] │
              └                               ┘
```

Figure 3.4: Fault matrices for a 5x5 torus-connected mesh and the *star* communication pattern.

IV. Providing Maximal Fault Tolerance

Determining the CEs and NFGs of the different processors finds the largest collection of component failures within a topology such that the algorithm can still detect all errors induced by these failures. We stated originally that we will consider algorithms that can tolerate up to t_g faults in each CE. We now define the minimum and maximum number of faults that can be tolerated simultaneously in an arbitrary topology using an error-detecting algorithm.

Trivially, the *minimal fault index* of a topology with respect to an algorithm that is able to tolerate t_g local faults is t_g, the local fault measure.

Definition: The *maximal fault index* (MFI) of a topology with respect to an algorithm that is able to tolerate t_g local faults is the number of failures t_s that can occur such that

$$(\forall i,\ 1 \le i \le n)(|FG(P_i)| \le t_g \wedge |\cup FG(P_i)| = t_s \text{ is maximal }).$$

Definition: The *fault tolerance decision problem* (FTD) determines if a total of t_s faults can be tolerated. It specifically checks the assignments for the different processors to give an answer to the following question:

For a given t_g and t_s, does there exist an assignment of FGs such that

$$(\forall i, 1 \le i \le n)(|FG(P_i)| \le t_g \wedge |\cup FG(P_i)| \ge t_s).$$

The solution of the FTD will depend on the topology as well as the communication pattern used in the algorithm. As in the matrix representation, we use a logical representation for faulty and non-faulty processors. Each row in the fault matrix represents a logical expression where "faulty" has the value **T**, "non-faulty" has the value **F**, and the "don't care" terms are not mentioned in the term. Thus, for example, the first row in Figure 3.4, which provides $NFG(P_1)$ in a 5x5 torus-connected mesh for $t_g = 1$, corresponds to

$$P_1 \wedge \overline{P_2} \wedge \overline{P_3} \wedge \overline{P_4} \wedge \overline{P_5} \wedge \overline{P_6} \wedge \overline{P_7} \wedge \overline{P_{10}} \wedge \overline{P_{11}} \wedge \overline{P_{16}} \wedge \overline{P_{21}} \wedge \overline{P_{22}}$$

This statement must be true if we know that P_1 is faulty and we can only tolerate one fault per CE, to guarantee that the algorithm can detect all errors caused by the faulty processor.

To solve the FTD of an arbitrary topology for a fixed t_g and t_s, we essentially want to determine if there exists a set of t_g terms represented by the rows of the fault matrix that can be true simultaneously. In the example given above for the 5x5 mesh and $t_g = 1$, if P_1 is faulty, another possibly faulty processor could for example be P_8, since the entry in the row that indicates $NFG(P_1)$ is a "don't care". In the next step we then evaluate how the faultiness of P_8 influences where other faulty processors may be located.

To determine if the assignments of truth values to the processor states permits the detection of all errors, we need to show that the NFGs of all faulty processors match, i.e., the conjunction of all processor states as indicated in the corresponding rows of the fault matrix must be true for the rows of all sets of faulty processors **P**. We therefore need to check the rows in the appropriate fault matrix where

$$(\forall P_i \in \mathbf{P})(P_i = \mathbf{T} \rightarrow (\forall j)(\wedge_{P_j \in NFG(\mathbf{P})} P_j = \mathbf{F}))$$

The time needed to determine if this is possible, for a specific assignment of logical values, is $O(n^2)$, i.e. polynomial. To then solve the FTD we check all possible 2^n assignments and evaluate each one of them to select the one(s) which permit the number of simultaneously faulty processors to be t_s.

A non-deterministic algorithm could now simply guess a correct assignment if we want to determine whether the FTD of a certain topology is equal to t_s for a fixed t_g. Because such a non-deterministic algorithm exists, we know that determining the FTD of an arbitrary topology is in NP.

Lemma 4.1: The FTD problem is in NP.

Proof: Using a non-deterministic algorithm we can find an assignment to the processors in polynomial time (see Theorem 3.3) that can tell if the FTD provides a result such that the MFI is equal to some value t_s. Thus FTD is in NP. □

Lemma 4.2: A variant of the 0,1 integer programming problem in which all components of $\vec{y}$ are required to be in {0,1}, called 0,1-integer programming, which is NP-complete, even if all components of each $\vec{x}$, b and all components of $\vec{c}$ are required to be in {0,1} [GaJo79], can be reduced to the FTD problem in polynomial time.

Proof: The integer programming problem consists of a finite set X of pairs $(\vec{x}, b)$, where $\vec{x}$ is an m-tuple of integers and b is an integer. We also have an m-tuple $\vec{c}$ of integers and an integer B. Integer programming solves the question whether there exists an m-tuple $\vec{y}$ of integers such that $\vec{x} \cdot \vec{y} \le b$ for all $(\vec{x}, b) \in X$ and such that $\vec{c} \cdot \vec{y} \ge B$. [2] A variant in which all components of $\vec{y}$ are required to be in {0,1}, called 0,1-integer programming, is still NP-complete, even if all components of each $\vec{x}$, b and all components of $\vec{c}$ are required to be in {0,1} [GaJo79]. In our particular case, b is only required to be non-negative since it will be used to indicate the number of faults which can be tolerated in each communication environment.

In the FTD, we have a set of n vectors, where processors in $CE(P_i)$, $1 \le i \le n$, have 1-coefficients in the vector. Others in the vector have 0-coefficients since they correspond to the elements outside the CE. This now makes up the finite set X of pairs $(\vec{x}, b)$ that is described above, where b corresponds to t_g, the maximal number of faults that can be tolerated in a CE and $\vec{x}$ marks the set of processors in the CE of the specific P_i we are considering. The value B described above gives the maximal number of faults that a topology can tolerate with respect to a certain communication pattern. This is the result of the FTD, and B corresponds to the value that we described so far by the variable t_s.

2: The dot-product $\vec{u} \cdot \vec{v}$ of two m-tuples $\vec{u} = (u_1, u_2, \ldots, u_m)$ and $v = (v_1, v_2, \ldots, v_m)$ is given by $\sum_{i=1}^{m} u_i v_i$.

We would now like to determine whether there exists a tuple $\vec{y}$ such that $\vec{x} \cdot \vec{y} \leq b$ for all CEs. This means that by determining $\vec{y}$ we assign a value of 0 or 1 to the processors to indicate whether they are non-faulty or faulty. $\vec{y}$ thus corresponds to the status of the processors and is therefore also an n-tuple. To translate the value of $\vec{y}$ into a processor status of faulty or non-faulty, we know that $\vec{y}$ can only be 0 or 1. Define the indicator I_A as

$$I_A = \begin{cases} 1 & \text{if } A \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

The following expression describes the bound on the number of local faulty components

$$\text{for each } P_i \ \sum_{j=1}^{n} I_{P_j \in CE(P_i) \wedge P_j \text{ is faulty}} \leq t_g$$

The vector $\vec{c}$ now shows which of the values in $\vec{y}$ we would like to select to compute the value of B. Since $\vec{y}$ gives the status of all processors, to find the maximal number of possible faulty processors, we need to consider all elements in $\vec{y}$, and thus the entries in $\vec{c}$ will all be 1.

By transforming each of the n CEs into {0,1}-vectors with n elements each, one for each of the n processors, and by expressing the maximal number of faults that can be tolerated by each CE as the pair $(\vec{x}, t_g)$, we obtain a 0,1-integer programming problem.

On the other hand, every 0,1-integer programming problem with the constraints listed as before, is also an FTD of an arbitrary topology with arbitrary communication patterns.

From the discussion we can see that the 0,1-integer programming problem can be reduced to the FTD problem in polynomial time. □

Lemma 4.3: Every 0,1-integer programming problem with $\vec{x}$ in {0,1} and non-negative b describes an FTD.

Proof: Each vector $\vec{x}$ of the 0,1-integer programming problem represents a communication environment where all 1s indicate processors within the CE, and all 0 entries are processors outside the CE. Then the ordered pair $(\vec{x}, b)$ indicates how many processors can be faulty in the particular CE. The value b thus corresponds to the local fault tolerance $t_{g'}$ which may vary for each CE. Solving a set of ordered pairs $(\vec{x}, b)$ for the {0,1} vector $\vec{y}$ will provide the assignments of 0s and 1s to the different processors which indicate faulty (1) or non-faulty (0) status of the processors, with each communication environment having at most $t_{g'} = b$ faults in each CE. □

As can be seen from the discussion above, 0,1-integer programming can be used to solve every FTD, and from Lemma 4.3, FTDs can be used to solve every 0,1-integer programming problem. Thus the two problems are equivalent, and therefore, since 0,1-integer programming is NP-complete, so must be the FTD.

Theorem 4.1: The FTD problem is NP-complete.

Proof: The proof follows directly from Lemmas 4.1 and 4.2. □

Corollary 4.1: The MFI problem is NP-hard.

Proof: To determine the maximal possible value of faulty components, we need to solve the 0,1-integer programming problem which is described by the FTD. This determines whether there exists a number of faulty processors $\geq t_s$, where t_s is an arbitrary integer $\leq n$, i.e., whether there are t_s 1-entries in the solution vector $\vec{c}$. We can thus solve the FTD at most (n-t_g) times to find the maximal value since t_g is the minimal fault index, and n is the theoretical maximum. Thus, MFI can be obtained from the FTD through a polynomial number of steps and is therefore NP-hard. □

Definition: A *processor group*, K, describes a collection of processes whose simultaneous failure still permits all errors caused by their failure to be detected. Processor groups can be mapped disjointly onto the actual processor topology.

Corollary 4.2: Partitioning of the individual processes onto processor groups based on an optimal fault distribution can be obtained in polynomial time from the solution of the MFI.

Proof: Instead of equating each process with its own processor, we now consider each process individually and try to partition all n processes onto a smaller number of m processor groups. We use the solution of the MFI which provides an optimal distribution of processor faults by providing the solution vector $\vec{c}$ for the fault matrix, indicating which processes may simultaneously be faulty. The NFG of each process indicates which other processes may not be located together on the same processor. The algorithm of Figure 4.1 provides a partitioning of the processes P_i to the processor groups K_j. This process is clearly polynomial. □

An example for the mapping is given in Section 5.1.2 for the star pattern.

```
/* All $K_l$ are processor groups to which the individual processes are mapped.
  The 1-entries in the solution vector $\vec{c}$ of the MFI are mapped to $K_0$,
  all remaining processes are mapped onto the other $K_l, i > 0$ */
$K_0 := \{\}$;
for $1 \le i \le n$ do
   if c[i]=1 then
         $K_0 := K_0 \cup i$;
         $P_i := mapped$;

/* Now distribute the remaining processes onto other processor groups: We cannot
  map a process into a group where it would be in the NFG of one of the other elements
  that are already mapped there. If no such group exists, a new one is created. */
$l := 1$;
$K_l := \{\}$;
for all $i \in K_0$
   for all $P_j \in CE(P_i) \wedge P_j \ne mapped$
      if $(\exists K_m, 1 \le m \le l)(\forall P_k \in K_m)(P_j \notin NFG(P_k)$ then
         $K_m := K_m \cup j$;
      else
         $l := l + 1$;
         $K_l := \{j\}$;
      end if
      $P_j := mapped$;
```

Figure 4.1: An algorithm to provide a mapping from the results of the FTD.

V. Finding The Maximal Fault Index For Fixed Topologies

Although determining an optimal distribution of faults is NP-hard for arbitrary graphs, this is not necessarily true for certain regular topologies and regular communication patterns. For example, in nearest neighbor algorithms, each processor and its neighbors form a communication environment. In these cases, it is easy to determine the maximal NFG overlap by inspection.

The topologies to be evaluated in this section are 2-dimensional torus-connected meshes and binary hypercubes. They provide the underlying interconnection network for error-detecting algorithms using regular communication patterns and $t_g = 1$. We will use compass coordinates to describe adjacency of processes.

5.1 MFI for Meshes

Because of the symmetry of the topology, we will focus our attention on torus-connected meshes only. The distribution of faulty components for meshes without wrap-around connections is similar but less restrictive since the wrap-around connections that guarantee the same number of adjacent processors to each node in the topology don't have to be considered.

5.1.1 Square Pattern

The first communication pattern evaluated here is communication in a "square". The communication environment for P is the set of processors P_E, P_S, and P_{SE}. This is shown in the left part of Figure 5.1.

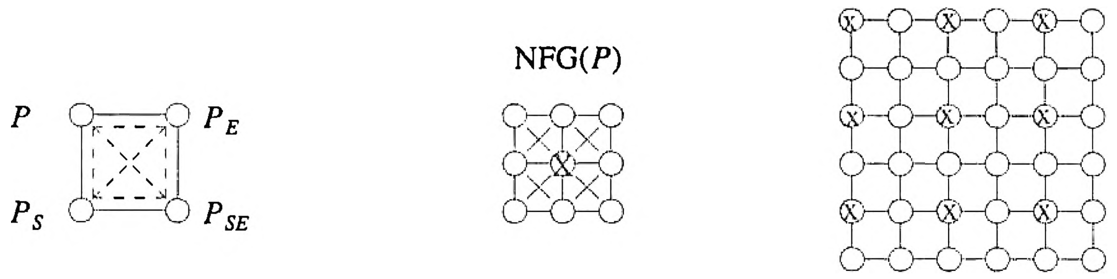


Figure 5.1: The "square" communication pattern, its NFG, and an optimal fault distribution in a torus-connected mesh (wrap-around not shown).

It can easily be seen from the augmented graph that P is also part of $CE(P_{NW})$, $CE(P_W)$ and $CE(P_N)$ due to adjacency. The pattern containing all processors in these CEs is, thus, a 3x3 processor group in which P is located at the center (see Figure 5.1, center). For $t_g = 1$ and P faulty, this is determined by the coloring algorithm of Section III as $NFG(P)$.

The maximal fault index places as many faulty processors as possible into the mesh. It is apparent, that for meshes smaller than 3x3 the MFI will be t_g. For an arbitrary $n \times m$ torus-connected mesh, with $t_g = 1$, the MFI can be determined by

$$MFI = div(m, 2) * div(n, 2)$$

which indicates the maximal possible number of faulty processors dependent on the number of rows and columns in the mesh. From Figure 5.1 one can see that all faulty processors must be at least a distance of two away from a known faulty processor. Since P is, optimally, exactly two away from the closest faulty neighbor, we can place up to $div(m, 2)$ into every other row and up to $div(n, 2)$ into every other column, which will give the result indicated above. A particular distribution is given in Figure 5.1 (right). Of course, if a larger number of processors is available, the processes on the 4 processors can be divided and placed onto the additional processors.

Partitioning the individual processes onto a smaller set of processors for the square pattern has already been shown as an example in Section I, Figure 1.2. The minimum number of processors required is 4, and the partitioning is obtained by placing non-overlapping CEs over the set of all processes, as described in Figure 1.2.c.

5.1.2 Star Pattern

Communication with all neighbors is also a common pattern for many algorithms. In this case, a processor P will communicate with P_E, P_W, P_N, and P_S. The augmented CE is shown in Figure 5.2 (top left). We will discuss this pattern again in Section VI for the evaluation of a relaxation algorithm.

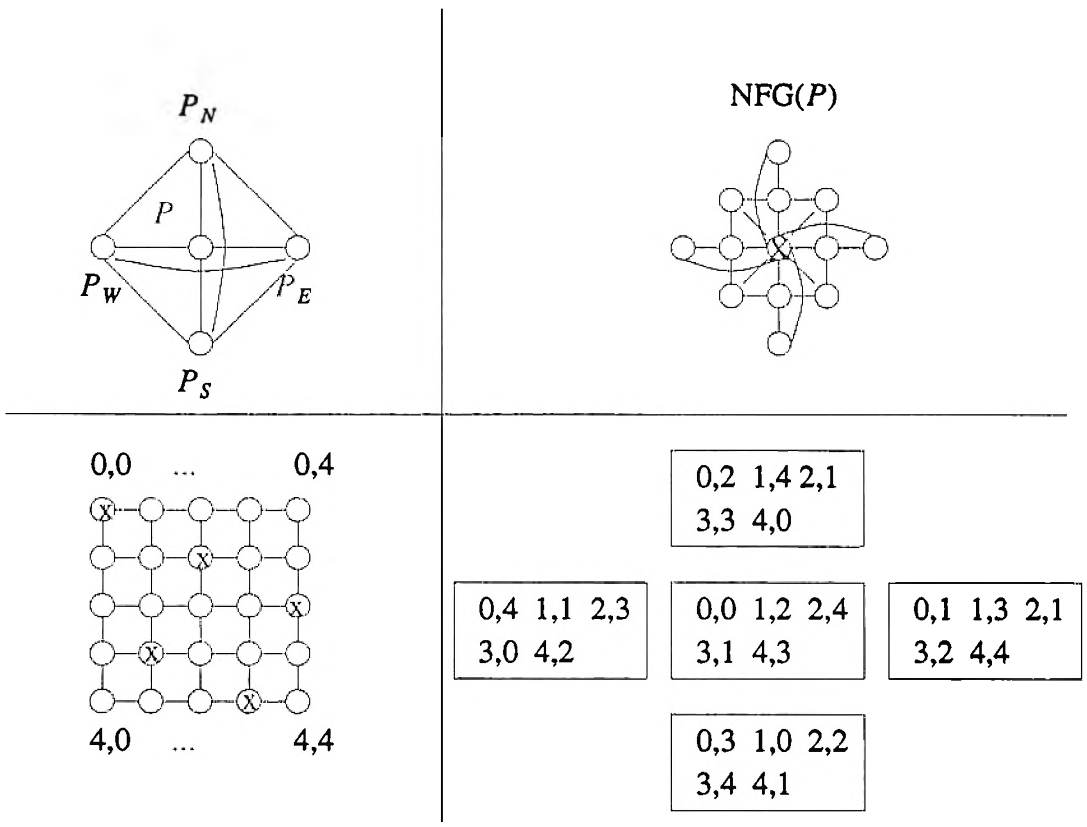


Figure 5.2: The "star" communication pattern, its NFG, an optimal fault distribution, in a torus-connected mesh (wrap-around not shown), and the fault-tolerant mapping.

As before, the goal is to permit as many faulty components as possible in the mesh but guaranteeing at the same time that each communication environment contains at most t_g faulty processors. To determine the NFG for each individually faulty processor we will again use the augmented graph and the coloring algorithm for finding the MFI.

We examine the case $t_g = 1$, where at most one fault can be tolerated in each CE. In this case the NFG for a faulty P as provided by the coloring algorithm on the augmented graph will result in a "star" pattern (Figure 5.2, top right). For P faulty and $t_g = 1$, none of these processors must be faulty.

In the ideal case we obtain a distribution of faulty processors that is identical to the *perfect 1-adjacency placement* of resources, where each non-resource node is adjacent to exactly one resource [RaCh92], which in our case is a faulty component. [RaCh92] show that the number of resource nodes in a k-ary n-cube for perfect 1-adjacency is

$$X = k^n/(2n+1), k > 2$$

which must be an integer. From this expression one can see that perfect 1-adjacency does not always exist, but it will nevertheless provide a bound on the number of faulty processors that can be permitted. A torus-connected mesh is a k-ary 2-cube, if we can guarantee that we have only $k \times k$ meshes. In this case the expression above becomes

$$X = k^2/5$$

which allows for up to 5 faulty components in a 5x5 torus-connected mesh. A possible distribution for this example is shown in Figure 5.2 (bottom left).

A fault-tolerant mapping for this particular communication pattern is given in Figure 5.2 (bottom right). Based on an optimal distribution of faults obtained earlier, the processes are placed such that only non-interfering processes are placed onto the same processor. The solution vector, $\vec{c}$, that was obtained from solving for the optimal fault distribution for this particular problem is [1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0], i.e., vertices (0,0), (1,2), (2,4), (3,1), and (4,3) in the problem graph can be simultaneously faulty.

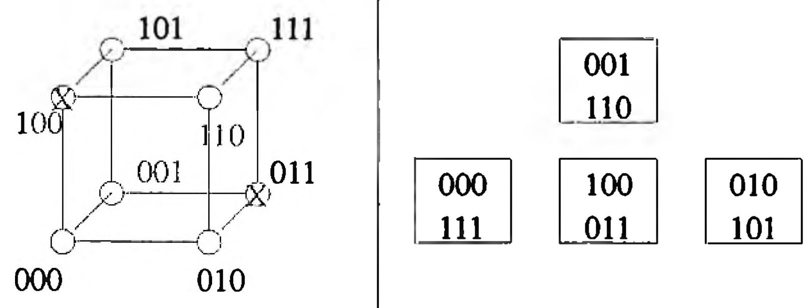
5.2 MFI for Binary Hypercubes

For binary hypercubes we are also frequently interested in communication patterns that communicate with adjacent processors only, i.e., into all dimensions of the hypercube. To determine the maximal fault index for this topology we use a similar approach as in Section 5.1. The problem becomes harder since the patterns formed by the NFGs are multi-dimensional and are therefore difficult to place by inspection, especially for high-dimensional hypercubes.

To obtain a star-like pattern, as described in Section 5.1.2, the faulty processors in the mesh as well as in the hypercube have to be at least a distance of three away from each other. In order to find a set of processors in the hypercube which all have this property, we can label the vertices of an n-dimensional hypercube in a binary gray code and then use Hamming codes to find the number of processors $B(n, d)$ which are a distance of d apart from each other. Specifically, for $d = 3$,

$$B(n,3) = 2^m \leq \frac{2^n}{n+1}$$

according to [Hamm50]. This provides an upper bound for the maximal fault index. An example for a 3-cube where a set of two faulty nodes which do not interfere with each others' computations and communications are marked is given in Figure 5.3 (left). A fault-tolerant mapping of the nodes onto a smaller set of processors is given in Figure 5.3 (right).



a) optimal fault distribution

b) fault-tolerant mapping based on the distribution

Figure 5.3: MFI and fault-tolerant mapping for nearest neighbor communication in a 3-cube.

VI. A Specific Example of an Error-Detecting Algorithm

Fault-tolerant algorithms can be generated using executable assertions for error detection [LuSM92, Andr79]. Executable assertions form logical tests which can be embedded into the program to verify, during the program execution, that the program meets its specifications. In general, we add executable assertions after each statement, which then verify that the previous statement was executed correctly, or which flag an error if the actual values of the variables do not match the range of the expected values. In case of an error, the assertions will force the program to halt execution.

For a specific problem and interconnection network, an error-detecting algorithm is able to handle a bounded number and particular distribution of failures. If this bound is exceeded or the distribution of faults is violated, the executable assertions may not be able to correctly detect all errors since multiple faults can mask each other.

In this section we discuss how the concepts described in the previous sections can be used to assess the fault tolerance of an error-detecting algorithm for matrix relaxation.

6.1 Iterative Relaxation

Iterative relaxation is one of the fundamental computation methods. Relaxation can be used in such diverse problem ranging from relaxation labeling [HuZu83] in distributed scene analysis to computational partial differential equation solvers [McNi88]. We present the general problem as approximating a solution to a large sparse system of linear equations $A\mathbf{u} = \mathbf{v}$, where $A = (a_{i,j})$ is a nonsingular $Q \times Q$ complex matrix, $\mathbf{v} = (v_i)$ is a complex vector, and $\mathbf{u} = (u_i)$ is the solution vector for $i, j \in \{1, 2, \ldots, Q\}$ and Q a perfect square. The method of Gauss-Seidel Relaxation is an iterative technique used to obtain an approximate solution, $\mathbf{u}^{(K)} = (u_i^{(K)})$, where K is the final iteration, to this system. The desired topology of the interconnection network for this computation is a two-dimensional mesh. The data exchange pattern for this algorithm corresponds to a communication with all adjacent processors in the mesh, which we described in Section 5.1.2 as the star pattern.

6.2 Error-Detecting Matrix Relaxation

Using Changeling [LuSm92], a program verification proof outline based on axiomatic semantics [3] is used to construct an error-detecting matrix relaxation. For the purposes of this

3: Axiomatic semantics provide formal statements about the effect of executing a propaper, we choose to concentrate on only one assertion from the matrix relaxation algorithm, which shows that at some final iteration step K, we have actually solved the original problem and found a solution. Simply put, this (post)assertion appears as

$$(A\mathbf{u}^{(K)} = \mathbf{v} - \varepsilon) \wedge (\forall i)(|u_i^{(K)} - u_i^{(K-2)}| < \varepsilon)$$

which ensures that the result obtained has converged on all nodes to within the desired tolerance ε. If the problem was solved correctly then the post assertion must hold; otherwise an error occurred which must be flagged.

The distributed program runs in two phases: in the first phase an iterative algorithm converges to a possible solution. Then a second phase, the verification of the solution, is used to check whether the post assertion is satisfied for all processes, i.e., whether the solution meets the desired specifications. If it does not, then we know that a fault must have occurred during the computation or during the verification process, indicating that the result cannot be trusted.

At the end of the final iteration K of the relaxation algorithm, the final result $u_i^{(K)}$ must satisfy the following relation:

$$\text{For } i \in \{1,2,\ldots Q\}, \quad \left| \frac{1}{a_{i,i}} \left[\sum_{j!=i}^{Q} a_{i,j} u_j^{(K)} - v_i \right] - u_i^{(k)} \right| \le \frac{\varepsilon}{\omega}$$

To verify the post assertion, each process will send its last computed value of $u_j^{(K)}$ to the other members of its CE using message diffusion [4] [CrAS85]. By checking the different versions that arrive on these paths [LuSM92], each processor in the CE must receive identical versions of a sent message or will detect an error if inconsistencies between messages from the same sender are discovered.

The system of equations to be solved by the relaxation algorithm has a unique solution. If two faulty processors in the same CE cooperate to fool the other processors then a spurious solution may be introduced which does not provide a correct solution to the problem but which cannot be detected. For example, consider solving the Laplace equation $\frac{\partial u^2}{\partial^2 x} + \frac{\partial u^2}{\partial^2 y} = 0$. A solution

gram. Assertions are made about program variables before, during and after program execution.

4: Message diffusion uses node-disjoint paths for sending at least two messages to the same destination, which can then be compared for consistency.

for this can be obtained by solving $u_{i,j} = \frac{1}{4}\,[u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}]$ which corresponds to the rows in a sparse matrix. What we actually verify in the postcondition is that for each CE the following relation between the local values of its components will be satisfied:

$$|u_{i+1,j} + u_{i-1,j} + u_{i,j+1,} + u_{i,j-1} - 4u_{i,j}| \le \varepsilon$$

For example, $u_{i,j} = 2$, $u_{i,j+1} = 2$, $u_{i,j-1} = 1$, $u_{i+1,j} = 4$ and $u_{i-1,j} = 1$ with $\varepsilon = 0.1$ satisfies this condition when locally tested in the CE. It is easy to see that a single component with a faulty value that violates the bound can always be detected. However, two faulty components can be faulty such that their errors add up without violating the bound (for example, $u_{i,j-1} = 0.5$ and $u_{i,j+1} = 2.5$), or they could cooperate by switching their values. If the components are not forced to use the same value in the verification round for all CEs in which they participate, then they could provide a correct value for the CEs in which they are the only faulty component and cooperate with another faulty component in the ones in which more than one is faulty. The CE for this example is shown in Figure 6.1. Thus, the verification round of the algorithm allows for $t_g = 1$, i.e. every single error in a CE can be detected.

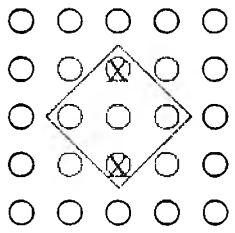


Figure 6.1: The Star Pattern with 2 cooperating errors in the same CE.

The actual communication pattern used in this matrix relaxation is an extended form of the star pattern to allow for message diffusion by providing node-disjoint paths from P to the components in the corners (Figure 6.2). Since the assertions can reliably detect up to one fault in each CE ($t_g = 1$), the upper bound on the number of faults that are permitted, the MFI, in a $Q \times Q$ mesh can be calculated as $Q^2/9$. Note that many different distributions of the faulty components are possible, as long as the condition of at most one faulty component per CE is not violated.

A possible fault-tolerant mapping is very similar to the one described in Section I as an example. We have 9 processor groups and map the individual processes according to (using an

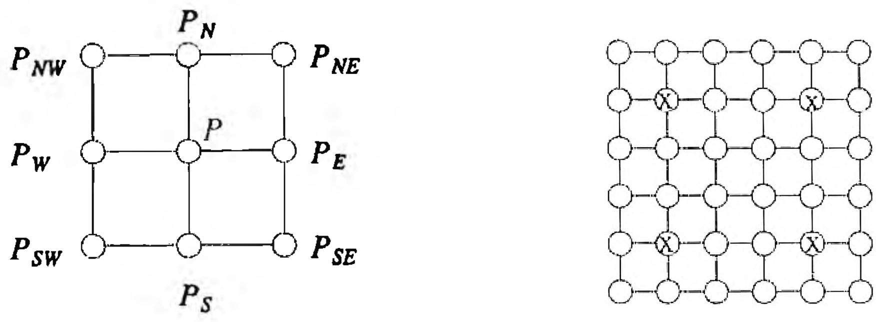


Figure 6.2: The communication pattern for message diffusion and a possible fault distribution.

x-y coordinate system): $Process_{i,j}$ maps into group $K_{l,m}$ if $i \; mod \; 3 = l \wedge j \; mod \; 3 = m$, assuming that we have no wrap-around connections.

VII. Conclusion

In this paper, the maximal fault index was introduced to demonstrate how a maximal number of simultaneous component failures can be tolerated by an error-detecting algorithm, based on specific distributions of the faults within the interconnection network. Depending on individual or sets of component failures, the non-fault groups of these components indicate where non-faulty components have to be located for the system to be able to detect all errors.

Although solving the maximal fault index problem for an arbitrary network topology and communication pattern is NP-hard, bounds are given in this paper for specific, frequently used communication patterns and topologies.

Based on the "optimal" distribution of faults, a partitioning technique can be used to assign processes to the processor groups in the system such that processes that may become faulty simultaneously, without their errors being able to mask one another, are located in the same processor group. These groups can then be mapped, disjointly, into the actual processor topology. Thus, the failure of a single processor will still allow for the detection of all errors.

The assessment of an error-detecting algorithm based on the concept of its minimal and maximal fault index can be used for safety critical systems, especially with respect to the fault-tolerant process-to-processor mapping that can be obtained from it. It will ensure that the failure of a single component does not go undetected, which increases the dependability of the system.

BIBLIOGRAPHY

[Andr79] Andrews, D.M. "Using executable assertions for testing and fault tolerance," *Proc. of the 9th FTCS*, 1979, pp. 102-105.

[CrAS85] Cristian, F., Aghili, H., and Strong, R. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Proc. of the 15th FTCS*, 1985, pp. 200-206.

[DaSe86] Dally, W. and Seitz, C. "The Torus Routing Chip," *Journal of Distributed Computing*, Vol. 1, No. 3, pp. 187-196, 1986.

[FoRa85] Fortes, J. and Raghavendra, C. "Graceful Degradable Processor Arrays," *IEEE Trans. On Computers*, Vol. C-34, Nov. 1985, pp. 1033-1044.

[GaJo79] Garey, M., and Johnson D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[GuRR93] Gu, D., Rosenkrantz, D.J., and Ravi, S.S. "Determining Performance Measures of Algorithm-Based Fault-Tolerant Systems," *J. of Parallel and Distributed Comp.*, Volume 18, No. 1, pp. 56-70, 1993.

[Hamm50] Hamming, R.W. "Error Detecting and Error Correcting Codes," *Bell Syst. Tech. J.*, Vol. 29, Apr. 1950, pp. 147-160.

[Haye76] Hayes, J. "A graph model for fault-tolerant computing systems," *IEEE Trans. On Computers*, Vol. C-25, Sept. 1976, pp. 875-883.

[HuZu83] Hummel, R. and Zucker, S., "On the Foundations of Relaxation Labeling Processes," *PAMI*, Vol. PAMI-5, No. 3, May 1984, pp. 267-287.

[JoAb87] Jou, J. and Abraham, J. "Fault-Tolerance Matrix Arithmetic and Signal Processing on Highly Computing Structures" *Proc. IEEE*, Vol. 74, No. 5, May 1986, pp. 732-741.

[LeLe85] Leighton, T. and Leierson, C. "Wafer-scale Integration of Systolic Arrays," *IEEE Trans. On Computers*, Vol. C-34, May 1985, pp. 448-461.

[LuSM92] Lutfiyya, H., Schollmeyer, M., and McMillin, B. "Fault-Tolerant Distributed Sort Generated from a Verification Proof Outline," *2nd Responsive Systems Symposium*, Springer-Verlag.

[McMi88] McMillin, B. "Reliable Parallel Processing: The Application-Oriented Paradigm," *Ph.D. Dissertation*, Michigan State University, 1988.

[McNi88] McMillin, B. and Ni, L., "Executable Assertion Development for the Distributed Parallel Environment," *Proc. of the 12th Int. COMPSAC*, October 1988, pp. 284-291.

[McNi92] McMillin, B. and Ni, L., "Reliable Distributed Sorting Through The Application-oriented Fault Tolerance Paradigm," *IEEE Trans. On Parallel and Distributed Comp.*, 1992, Vol. 3, No. 4, July 1992, pp. 411-420.

[RaCh92] Ramanathan, P. and Chalasani, S. "Resource Placement in k-Ary n-Cubes," *Proc. Intern. Conf. on Parallel Processing*, 1992, pp. II-133-140.

[Rose83] Rosenberg, A. "The Diogenes Approach to testable Fault-Tolerant Arrays of Processors," *IEEE Trans. On Computers*, Vol. C-32, Oct. 1983, pp. 902-910.

[ScMc93] Schollmeyer, M., and McMillin, B. "A General Method for Maximizing the Error-Detecting Ability of Distributed Algorithms," *UMR Technical Report*, Dept of Comp. Science, CSC-93-16.