INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 8907010

Distributed routing and task allocation in multicomputer systems

Chen, Ming-Syan, Ph.D.

The University of Michigan, 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

DISTRIBUTED ROUTING AND TASK ALLOCATION IN MULTICOMPUTER SYSTEMS

by

Ming-Syan Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Information and Control Engineering)
in The University of Michigan
1988

Doctoral Committee:

Professor Kang G. Shin, Chairman
Assistant Professor Chaitanya K. Baru
Assistant Professor Yung-Chia Lee
Assistant Professor Tim McLarnan
Associate Professor Quentin F. Stout

RULES REGARDING THE USE OF

MICROFILMED DISSERTATIONS

Microfilmed or bound copies of doctoral dissertations submitted to The University of Michigan and made available through University Microfilms International or The University of Michigan are open for inspection, but they are to be used only with due regard for the rights of the author. Extensive copying of the dissertation or publication of material in excess of standard copyright limits, whether or not the dissertation has been copyrighted, must have been approved by the author as well as by the Dean of the Graduate School. Proper credit must be given to the author if any material from the dissertation is used in subsequent written or published work.

For my parents

ACKNOWLEDGEMENTS

I would like to express my deepest thanks to my advisor Prof. Kang G. Shin. The constant guidance, timely encouragement and unwavering support he provided me to pursue this research has been priceless during my entire graduate study.

Many thanks are extended to the members of my committee for their constructive criticism and time spent in reviewing this dissertation. Also, I am obliged to the Office of Naval Research and National Aeronautics and Space Administration for their financial support (ONR contract N00014-85-k-0122 and NASA grant NAG-1-296).

I am forever grateful to my parents for the support and encouragement they have given me that made my graduate study possible. Finally, a very special thank goes to my fiancee Jeng-Chun, with whom I feel very fortunate to be able to share my life, for her encouragement and confidence in me.

TABLE OF CONTENTS

LIST OF FIGURES

Figures

LIST OF TABLES

Tables

CHAPTER 1

INTRODUCTION

1.1. Overview

In recent years, the advances of VLSI and computer networking technologies have made it attractive to use multicomputer systems for many applications. Specifically, by multicomputer systems, we mean loosely coupled MIMD (Multiple Instrunction Stream - Multiple Data Stream) machines [30]. Such multicomputer systems not only provide the capability of utilizing remote computer resources or data which are not available at a local computer node but also, as compared with a conventional SISD (Single Instrunction Stream - Single Data Stream) machine, improve the throughput of the whole system by executing several tasks in parallel [18, 23, 92]. Owing to their efficiency, flexibility and reliability, multicomputer systems have drawn a significant amount of research effort [55, 68, 77, 83, 93]. Routing and task allocation, among others, are key factors to the performance of a multicomputer system. Instead of using shared memory, all communication and synchronization between two computer nodes is achieved via message passing, which, in turn, relies heavily on an efficient routing scheme. On the other hand, to utilize the resources in a multicomputer system effectively, it is essential to have a task allocation strategy which can reduce inter-processor communication as well as exploit the the inherent parallelism during task execution. For the reasons above, development of efficient strategies for routing and task allocation in multicomputer systems is taken as the main objective of this dissertation.

In Chapter 2, we shall investigate the routing in wide-area computer networks. Various control strategies for routing have been proposed [6, 9, 59, 64, 88]. Non-adaptive routing strategies, such as random routing and fixed routing, make no attempt to adjust themselves to changes in network conditions and are found to be too inefficient and unreliable. Centralized adaptive routing strategies, which use a routing control center to keep information on the entire network and make the routing decisions for individual nodes, suffer from vulnerability to single point failures and the need for a routing control center [9]. On the other hand, distributed adaptive routing strategies distribute network information among all computer nodes and, thus, avoid the disadvantages of the other strategies [58]. Among all distributed adaptive routing strategies reported in the literature, ARPANET's previous routing strategy (APRS), which uses a network delay table for making routing decisions, appears to be acceptable for most packet switching networks [60]. However, this routing strategy has been replaced by ARPANET's current routing strategy (ACRS) [61], due to its poor adaptability to network changes and unnecessary looping in case of network failures, i.e., packets are returned to a node from which they were previously transmitted [61, 66].

The reduction of looping effects is the subject of Chapter 2. We shall show that the network's adaptability is improved by incorporating more information in routing messages. It will be shown that a routing strategy can eliminate multi-node loops by keeping in network delay tables not only the delay of a minimal path to every other node but also a set of first few nodes in the path. The number of nodes included in the routing message is referred to as the *order* of the routing strategy. We shall show that as the order of a routing strategy increases, longer loops will be eliminated. Moreover, depending on the network structure, we can determine the order of a routing strategy required for *each* node in order to eliminate looping completely. Unlike the other distributed routing strategies where the same strategy is applied indiscriminately to every node in a network, the order of a node's routing strategy

depends on the network topology and may vary from one node to another.

It can be seen that the results in Chapter 2 are applicable to computer networks of arbitrary topology. Notice, however, that multicomputer interconnection networks usually have regular topological structures so as to exploit the benefits from the inexpensive replication of hardware components as well as the applicability of identical software and protocols to each computer node. Clearly, the regularity of the network topology will enable each node to determine its path to every other node for message routing without using network delay tables. In view of this fact, in Chapter 3 we shall focus our attention on the routing in multicomputer interconnection networks.

Various research and development results on how to interconnect multicomputer components have been reported in the literature [2, 25, 28, 76, 82]. Some of them have been compared with each other [26, 93]. However, most of them are not satisfactory due mainly to their inability of providing the following features: the simplicity of interconnection, the efficiency of message routing and broadcasting, a fine scalability, and a high degree of fault-tolerance.

Hexagonal meshes (or H-meshes) are presented as a candidate multicomputer architecture in Chapter 3. A large number of data manipulation applications require the processing nodes on the hexagonal periphery to be wrapped around to achieve regularity and homogeneity. From a topological point of view, the way of wrapping determines the type of H-mesh. To the best of our knowledge, there is no general efficient method for wrapping, routing and broadcasting in H-meshes known to date. Consequently, in Chapter 3, we shall first present a systematic method for wrapping H-meshes, and then propose a new, simple addressing scheme for the wrapped H-meshes. Efficient routing and broadcasting algorithms under the new addressing scheme will also be developed. As we shall see, the proposed addressing

scheme will achieve the homogeneity of the network and facilitate routing and broadcasting in the H-meshes significantly.

Recently, the increasing use of multiprocessor/multicomputer systems for reliability-critical applications has made it important to design fault-tolerant routing strategies for such systems. Therefore, in Chapter 4 we shall concern ourselves with fault-tolerant routing in multicomputer systems. Two fault-tolerant routing schemes, based on depth-first search and network delay tables respectively, will be developed for the routing in hypercube multicomputers.

A connected hypercube with faulty components (nodes or links) is called an *injured hypercube*, whereas a hypercube without faulty components is called a *regular hypercube*. It is well-known that routing in a regular hypercube can be handled by a systematic procedure [72]. However, this scheme becomes invalid in an injured hypercube, since the message may be routed to a faulty component. In order to enable non-faulty nodes in an injured hypercube to communicate with one another, some network information must be kept in either the message itself or each node so that the route chosen can bypass the faulty components.

Therefore, in Chapter 4 we shall first develop a routing scheme based on depth-first search, in which each node is required to know only the condition (healthy or faulty) of its own links. The network information is added to the message as the message travels toward the destination. However, due to the absence of network information at each hypercube node, the paths chosen by the above scheme may not be the shortest. Clearly, this routing scheme, though avoids the necessity of keeping network information at each node, is not preferred when two nodes communicate with each other frequently, since in such a case extra hops have to be traversed during each transmission. In light of the idea developed in Chapter 2, a routing algorithm using network delay tables to keep network information will be developed to

remedy the foregoing drawback. Observe that the abundant connections in a hypercube usually make routing decisions in a node unaffected by the failure of a component located far away from the node. This implies that the network delay table of each node does not have to keep information on those nodes located far away, and can still lead to the shortest path routing.

We begin our study of the task allocation in multicomputer systems in Chapter 5. In a multicomputer system, a task is usually decomposed into a set of cooperating task modules which are assigned to a set of processors in order to exploit the inherent parallelism during task execution [18, 19, 77]. Each task can be described by an undirected graph called the *task graph*, $G_T = (V_T, E_T)$, where V_T is the set of nodes, each representing a task module, and $E_T \subseteq V_T \times V_T$ is the set of edges, each representing the need of communication between task modules. When there is an edge between two task modules in G_T, the two modules are said to be *related* to each other. Similarly, a multicomputer system can be represented by an undirected graph called the *processor graph*, $G_P = (V_P, E_P)$, where V_P is the set of processors in the system and $E_P \subseteq V_P \times V_P$ is the set of edges representing communication links between processors.

The maximal number of hops between two processors to which two related task modules are assigned is called the *dilation* of that assignment [39]. Cooperating task modules are usually required to be assigned to a set of processors in such a way that the communication delay between any two related task modules is kept low. One way to accomplish this is to limit the dilation to a small number so that two related task modules are assigned to processors located physically close to each other. An assignment is said to be *acceptable* if its dilation is less than or equal to a prespecified integer value.

The problem of deriving an "optimal" (in the sense of, for example, load balancing or minimization of job execution time) task assignment is known to be very difficult [69, 31]. In [77] the task assignment problem is formulated as a state-space search problem which is then solved by the A* algorithm [65]. However, without the knowledge of the number of acceptable assignments for given G_T and G_P, one cannot tell the size of the state-space to be searched in the A* algorithm. Note that such an algorithm often requires a large number of evaluations of a complex heuristic function. As will be discussed later, the knowledge of the number of acceptable assignments can be used not only for providing a simplified state-space search but also for reducing the size of the state-space to be searched. Therefore, in Chapter 5, we shall address how to obtain the number of acceptable task assignments for given G_T and G_P. It will be seen that the combinatorial approach we shall use in Section 2.2.2 to analyze the complexity of the optimization procedure is a special case of the results in Chapter 5.

In Chapter 6, we study the task allocation problem in hypercube multicomputers. A task arriving at a hypercube multicomputer must be assigned to a subcube of the size required for execution of the task. Upon completion of execution, the subcube used for the task must be released for later use. As it will become clear later, there is a close resemblance of the task allocation problem in hypercube multicomputers to the conventional memory allocation problem. In both problems, we want to maximize the utilization of available resources and also minimize the inherent system fragmentation.

The strategy currently used in the NCUBE machine, called the *buddy strategy*, will be investigated first. We shall show that due to the special structure of a hypercube, the availability of some subcubes cannot be detected by the buddy strategy, and the processor utilization is thus degraded. A processor allocation strategy using the binary reflected Gray code, called the *GC strategy*, is then described and shown to be able to recognize more subcubes than the

buddy strategy can. The performances of both the buddy and GC strategies will be comparatively analyzed. Both strategies will be proven to be *statically optimal* in the sense that only minimal subcubes are used by both strategies to accommodate each sequence of incoming tasks when processor relinquishment is not considered. It will also be proven that subcube recognition ability is enhanced significantly by the GC strategy; the number of recognizable subcubes in the GC strategy is twice that in the buddy strategy. Furthermore, we shall show that the GC strategy is optimal in terms of the subcube recognition ability of a sequential search. As will be seen, there are many different GC's, each of which is associated with a set of recognizable subcubes. The subcube recognition ability can be improved by using more than one GC. The relationship between the GC's used and their subcube recognition ability will also be derived and used for determining an allocation strategy with multiple GC's.

Like the conventional memory management, a sequence of allocation and deallocation of subcubes may result in a fragmented hypercube, where a sufficient number of unoccupied nodes exist but do not form a subcube large enough to accommodate an incoming task. Such fragmentation leads to poor utilization of hypercube nodes. As the fragmentation problem in the conventional memory allocation can be handled by memory compaction, the fragmentation problem in a hypercube can be solved by *task migration*, i.e., relocating and compacting active tasks within the hypercube. Consequently, the procedure for the task migration under the GC strategy is developed in Chapter 6. We shall derive the goal configuration, the node-mapping between the source and destination subcubes, and a routing scheme which leads to the shortest deadlock-free paths for task migration.

1.2. Outline of the dissertation

This dissertation is organized as follows. Distributed adaptive routing strategies for wide-area computer networks are treated in Chapter 2. In Chapter 3, we propose the routing and broadcasting schemes for hexagonal mesh multicomputers. Fault-tolerant routing schemes for hypercube multicomputers are developed in Chapter 4. In Chapter 5, by using the knowledge on the number of acceptable task assignments, we propose a new approach to reduce the difficulty of the task allocation problem in heterogeneous multicomputer systems. We develop task allocation strategies for hypercube multicomputers in Chapter 6. An associated task migration strategy is also presented. Concluding remarks are given in Chapter 7. A list of symbols used is given in the Appendix.

CHAPTER 2

DISTRIBUTED ADAPTIVE ROUTING IN COMPUTER NETWORKS

For computer networks, routing is very important to their performance and reliability [59,75]. Among the various routing algorithms proposed [10,45,60,61,62,64,78,86], distributed adaptive routing algorithms have drawn considerable attention because of their high potential for reliability and adaptability. The ARPANET's previous routing strategy (APRS) [60] is a typical example of these. Under APRS, the path from one node to every other node is not determined in advance. Instead, every node maintains a *network delay table* to record the shortest delay via each link emanating from the node. A *minimal delay table* in a node, which contains the delays of the optimal paths (i.e., the path requiring the minimal delay) from that node to all the other nodes is passed to all of its adjacent nodes as a routing message at every fixed time interval (e.g., 128 ms in APRS). Note, however, that under APRS each node sends the same routing message to all its neighbors without making any distinction between receiving nodes. This forces some nodes to receive useless routing messages, thereby resulting in undesirable looping in case of link/node failures. The process for each node to obtain new correct information for its network delay table after certain failures, which is termed as the network recovery process, will thus be delayed [66].

The routing algorithms proposed in [45,62,86] have the same major features as the one in APRS, except they employ more provisions to cope with network failures. However, they still cannot avoid some inherent drawbacks such as poor adaptability and inefficiency [45,46]. The ARPANET's current routing strategy (ACRS) [61] uses a different approach for handling

routing messages. In ACRS, every node in the network is required to keep and maintain information of the entire network. ACRS will always reach a correct routing decision as long as the global information at each node is accurate and consistent. However, this strategy requires every node to contain a large storage area for the global information and may make the entire network congested with messages for updating the global information.

The routing strategy in [78] as well as the one adopted in the TIDAS network [10] is similar to APRS except for the following modification. If the routing message is sent from node N_j to node N_i which is the second node in the optimal path from N_j to some other destination node N_d, the delay of the optimal path from N_j to N_d was replaced with the delay of its *second optimal path* in the routing message passed to N_i. Although this modification leads to a significant improvement over APRS in reducing the looping effects, it does not eliminate them completely. In [78], we have rigorously analyzed the performance of a routing strategy using the above modification. We proved that, although ping-pong type loops (i.e., loops with two nodes) can be removed from the network delay table of each node by the above modification, multi-node loops (i.e., loops with more than two nodes) may still exist. In this chapter, we shall extend our analytical results to routing strategies which are free of multi-node loops and show that a routing strategy can remove multi-node loops from the network delay table of each node by keeping in network delay tables not only the delay of each minimal path but also a set of first few nodes in the path. The number of nodes included in the routing message is referred to as the *order* of the corresponding routing strategy. The number of nodes in a loop that will be eliminated by a routing strategy increases with the order of the routing strategy [78].

Consider the following straightforward approach to remove looping completely from the network delay table of each node: All nodes in each path are included in routing messages

and sent to neighboring nodes. However, this naive approach is very inefficient due to its excessive overhead. Consequently, it is very important to determine the *minimal* order of routing strategy required for each node to make all the paths implied in its network delay table loop-free. As we shall prove later, depending on the network structure, the portion of a path that each node should keep and send to its neighboring nodes in order to eliminate looping completely can be determined. Unlike the other distributed routing strategies where the same strategy is applied indiscriminately to every node in a network, the order of the routing strategy required for each node depends on the network topology and may vary from one node to another. Notice that we remove looping effects by augmented minimal delay vectors, whereas the method described in [64] is based on the use of extensive protocols.

This chapter is organized as follows. In Section 2.2, we present necessary definitions and notation, and then introduce the high order routing strategies. Performance of the high order routing strategies will also be analyzed. In Section 2.3, we develop formulas to determine the minimal order of the routing strategy required for each node to remove looping from its network delay table completely. We shall take into consideration the operational overhead in handling routing messages and optimize the tradeoff between the network adaptability and the operational overhead. Complexity of the optimization algorithm is also analyzed. The results developed in this chapter have also been reported in [78, 79].

2.1. High Order Routing Strategies

In Section 2.1.1, we shall introduce notation and definitions required, which is followed by the description of the high order routing strategies in Section 2.1.2. Performance analysis of the high order routing strategies will be given in Section 2.1.3.

2.1.1. Preliminaries

For a computer network N, let $V(N)$ and $E(N)$ denote respectively the set of computer nodes and the set of computer links with $|V(N)| = p$ and $|E(N)| = q$, where $|S|$ represents the cardinality of the set S. To illustrate the network recovery process after link/node failures, we assume that the network N is connected after link/node failures throughout our discussion. Let $DL_{i,j}$ be the delay of a direct link $L_{i,j}$ from N_i to N_j. The set of nodes adjacent to N_i is denoted by A_i. There are usually many paths from N_i to N_j, which are represented by the set $SP_{i,j}$, and let $SP \equiv \bigcup_{N_i, N_j \in V(N)} SP_{i,j}$. Let $P_{i,j}$ denote a path with the shortest delay[1] (i.e., an optimal path) in $SP_{i,j}$, and $P_{i,j-u,v}$ be the shortest delay path from N_i to N_j without traveling on $L_{u,v}$. Clearly, $P_{i,j-u,v}$ is the new optimal path from N_i to N_j if the link $L_{u,v}$ becomes faulty. Note that $P_{i,j-u,v} = P_{i,j}$ only when $L_{u,v}$ is not a part of $P_{i,j}$.

A path in N is expressed by an *ordered sequence representation* of nodes. For example, a path $P_i \in SP$ can be represented by $(N_{i_0}, N_{i_1}, \cdots, N_{i_m})$. Let $H_k(P_i)$ be the set of the first k nodes of a path $P_i \in SP$. For a path $P_i = (N_{i_0}, N_{i_1}, \cdots, N_{i_m})$, $H_k(P_i) = \{N_{i_0}, N_{i_1}, \cdots, N_{i_k}\}$ if $m+1 \geq k$, and $H_k(P_i) = \{N_{i_0}, N_{i_1}, \cdots, N_{i_m}\}$ otherwise. In addition, a function $h : SP \rightarrow \mathbf{I}^+$ is the *hop function* of a path, where $h(P_i)$ denotes the number of links (hops) in a path $P_i \in SP$ and $\mathbf{I}^+$ the set of positive integers, and a function $d : SP \rightarrow \mathbf{R}^+$ is the *delay function* of a path, where $d(P_i)$ is the summation of all link delays in a path $P_i \in SP$ and $\mathbf{R}^+$ the set of positive real numbers. Then, the average link delay in path P_i can be denoted by a function $ave : SP \rightarrow \mathbf{R}^+$ such that $ave(P_i) = d(P_i)/h(P_i)$.

Definition 2.1: If a path $P_i = (N_{i_0}, N_{i_1}, \cdots, N_{i_m})$ is faulty and the pair $(N_{i_k}, N_{i_{k+1}})$, i.e., $L_{i_k,i_{k+1}}$, represents the first faulty link in the ordered sequence representation of P_i, then k is called the

[1]There may be many paths with the same shortest delay, in which case, an arbitrary one is chosen to break the tie.

screen of P_i and denoted by $scn(P_i)$. If the path P_i is not faulty, then $scn(P_i)=\infty$.

Definition 2.2: A *loop* is a path which starts and ends at the same node, and does not contain any other loop, i.e., a path $(N_{i_0}, N_{i_1}, \cdots, N_{i_m})$ is a loop iff $N_{i_0} = N_{i_m}$, and $N_{i_j} \neq N_{i_q}$ for $0 \leq j, q \leq n$ except $N_{i_j} = N_{i_q} = N_{i_0}$. Also, a loop L_i is called a *k-th order loop* if the number of hops in L_i is k+1, i.e., $h(L_i) = k+1$.

The set of loops starting and ending at N_j is denoted by $SL_{j,j}$. A path P_i is said to be *k-th order loop-free* iff there is no occurrence of any j-th order loop in P_i, for $1 \leq j \leq k$. Let SP^k, $k \in I^+$, be the set of paths which are k-th order loop-free and $SP^k_{i,j} \equiv SP^k \cap SP_{i,j}$ be the set of k-th order loop-free paths from N_i to N_j. Obviously, $SP^{k+1} \subseteq SP^k$, $\forall\, k \in I^+$. In addition, a routing strategy is said to be free of the k-th order loop if all the paths implied in the network delay table of each node are k-th order loop-free.

For example, $(N_2, N_3, N_2, N_3, N_2)$ is not a loop and (N_1, N_2, N_3, N_1) is a 2-nd order loop. For the path $P_i = (N_1, N_2, N_1, N_2, N_1, N_3)$ in the example network of Figure 2.1, $d(P_i) = 12$, $h(P_i) = 5$, $ave(P_i) = 2.4$, and P_i contains a 1-st order loop occurring between N_1 and N_2. Hence, $P_i \in SP_{1,3}$ but $P_i \notin SP^1_{1,3}$. When the link $L_{1,3}$ is faulty, P_i is faulty and $scn(P_i) = 4$.

2.1.2. Description of high order routing strategy

As we mentioned before, under APRS the path from one node to every other node is not determined in advance. Instead, every node maintains a network delay table to record the shortest delay via each link emanating from the node. When a node has to route a packet, it determines from its network delay table the next hop for the packet which will lead to the shortest delay path. A *minimal delay table* in a node, which contains the delays of the shortest paths from that node to all other nodes, is passed to all of its adjacent nodes as a routing

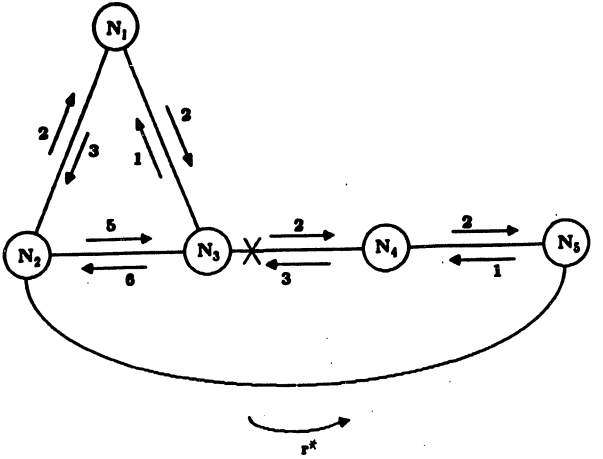


Figure 2.1. A computer network with link delay specified.

message every fixed time interval.

Let $NT.dly_{\overline{i,j},d}(m)$ denote the delay from N_i via N_j to N_d in the network delay table of N_i under the APRS during the time interval [m, m+1) and $DL_{i,j}(m)$ be the delay of the link $L_{i,j}$ at time m. Also, let $DOP_{i,d}(m)$ denote the shortest delay from N_i to N_d in the network delay table of N_i during the interval [m, m+1), and denote the corresponding path as $OP_{i,d}(m)$. $DOP_{i,d}(m)$ is then sent to all adjacent nodes of N_i as routing information. Then, the operations of the APRS can be formally described as follows.

$$NT.dly_{\overline{i,j},d}(m) = DL_{i,j}(m) + DOP_{j,d}(m-1) \quad \forall\ N_j \in A_i \tag{2.1}$$

$$DOP_{i,d}(m) = \min_{N_j \in A_i} \left\{ NT.dly_{\overline{i,j},d}(m) \right\}. \tag{2.2}$$

An example of the network recovery process under APRS for the network in Figure 2.2 is given in Table 2.1. Notice that after the failure of $L_{4,5}$ at time t=0, N_4, according to the routing message it received from N_2, thought there is a path with delay 4 via N_2 to N_5, which, however, is (N_4, N_2, N_4, N_5) and not available. When time t=2, N_4, according to information it collected thus far, knew the path with delay 4 is faulty, but still thought there is a path with delay 6, which is (N_4, N_2, N_4, N_2, N_4, N_5) and not available either. Eventually, it took 20 time intervals for N_4 to obtain a correct shortest delay to N_5. It can be verified that while misled by the routing message sent from N_4, nodes N_1, N_2 and N_3 needs 20, 19 and 17 time intervals respectively to obtain their new shortest delays to N_5, and the network recovery process is thus delayed. As it can be observed from Table 2.1, the slowdown of the network recovery process is mainly caused by a 1-st order loop, (N_4, N_2, N_4).

To remedy this, under the strategy we developed in [78] and the one in the TIDAS network, every node keeps a network delay table and exchanges minimal delay tables with all its adjacent nodes as in APRS except for the following modification. If the routing message is

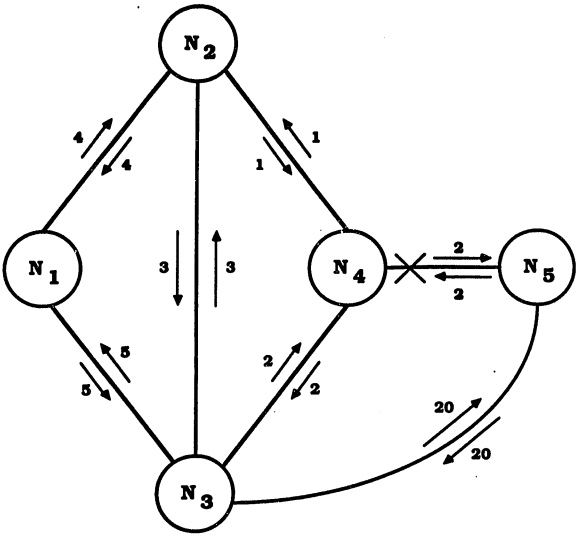


Figure 2.2. A network where $L_{4,5}$ is broken.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	$t=k,\ 4 \le k \le 15$	t=16	t=17	t=18	t=19	$t \in [20, \infty)$
N_2	7	7	7	9	9	$\lfloor \frac{n}{2} \rfloor 2+7$	23	23	25	25	27
N_3	9	9	9	11	11	$\lfloor \frac{n}{2} \rfloor 2+9$	25	25	25	25	25*

(a). Network delay table of N_1.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	$t=k,\ 4 \le k \le 15$	t=16	t=17	t=18	t=19	$t \in [20, \infty)$
N_1	11	11	11	11	13	$\lfloor \frac{n}{2} \rfloor 2+9$	25	27	27	29	29
N_3	7	7	7	9	9	$\lfloor \frac{n}{2} \rfloor 2+7$	23	23	23	23	23
N_4	3	3	5	5	7	$\lfloor \frac{n}{2} \rfloor 2+3$	19	21	21	23*	23

(b). Network delay table of N_2.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	$t=k,\ 4 \le k \le 15$	t=16	t=17	t=18	t=19	$t \in [20, \infty)$
N_1	12	12	12	12	14	$\lceil \frac{n}{2} \rceil 2+10$	26	28	28	30	30
N_2	6	6	6	8	8	$\lfloor \frac{n}{2} \rfloor 2+6$	22	22	24	24	26
N_4	4	4	6	6	8	$\lceil \frac{n}{2} \rceil 2+4$	20	22	22	24	24
N_5	20	20	20	20	20	20	20	20*	20	20	20

(c). Network delay table of N_3.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	$t=k,\ 4 \le k \le 15$	t=16	t=17	t=18	t=19	$t \in [20, \infty)$
N_2	4	4	4	6	6	$\lfloor \frac{n}{2} \rfloor 2+4$	20	20	22	22	24
N_3	6	6	6	8	8	$\lfloor \frac{n}{2} \rfloor 2+6$	22	22	22	22	22*
N_5	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

(d). Network delay table of N_4.

Table 2.1. Network delay tables of N_1, N_2, N_3 and N_4 under APRS where N_5 is the destination node.

passed from N_j to N_i which is the second node in the optimal path from N_j to some destination node N_d, we replace the delay of the optimal path from N_j to N_d with the delay of its second optimal path (i.e., the path requiring the second shortest delay to N_d among all paths in the network delay table of N_j) in the routing message passed to N_i. As it will be pointed out later, this routing strategy is a special case of the high order routing strategies and can be viewed as the 1-st order routing strategy. Let $NT.dly^1_{N_j,d}(m)$ denote the delay from N_i via N_j to N_d in the network delay table of N_i under the 1-st order routing strategy during the time interval [m, m+1). Then, the operation of the 1-st order routing strategy can be expressed as follows.

$$NT.dly^1_{N_i,d}(m) = DL_{i,j}(m) + \min_{N_q \in A_j,\ q \neq i} \left\{ NT.dly^1_{j,d}(m-1) \right\} \tag{2.3}$$

An example of the network recovery process for the network in Figure 2.2 under the above modification to APRS is given in Table 2.2. It can be seen that the time intervals required for N_1, N_2, N_3 and N_4 to determine their new optimal paths to N_5 become 11, 10, 8 and 9, respectively.

From Table 2.2, it can be seen that the 1-st order loops will be removed from the network delay table of each node by the 1-st order routing strategy, whereas loops with more than two nodes (i.e., multi-node loops) may still exist. Notice that the 1-st order routing strategy removes the 1-st order loops by disallowing each node to send its neighbors the routing messages which have been found to be useless to receivers. However, under the 1-st order routing strategy every node can determine only the delay and the second node of each path from its network delay table. Naturally, as the amount of local information is increased, every node is expected to form more useful routing messages leading to a greater reduction of looping effects. In fact, a routing strategy can be developed to remove multi-node loops from the

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	$t=8$	$t=9$	$t=10$	$t=11$	$t \in [12, \infty)$
N_2	7	7	7	11	13	13	17	19	19	23	25	25	27	27
N_3	9	9	9	11	15	15	17	21	21	23	25	25	25*	25

(a). Network delay table of N_1.

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	$t=8$	$t=9$	$t=10$	$t=11$	$t \in [12, \infty)$
N_1	13	13	13	13	15	19	19	21	25	25	27	29	29	29
N_3	7	7	7	13	13	13	19	19	19	23	23	23	23*	23
N_4	3	3	9	9	9	15	15	15	21	21	21	23	23	23

(b). Network delay table of N_2.

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	$t=8$	$t=9$	$t=10$	$t=11$	$t \in [12, \infty)$
N_1	12	12	12	12	16	16	18	22	24	24	28	30	30	32
N_2	6	6	6	12	12	12	18	18	18	24	24	24	26	26
N_4	4	4	10	10	10	16	16	16	22	22	22	26	26	26
N_5	20	20	20	20	20	20	20	20	20	20*	20	20	20	20

(c). Network delay table of N_3.

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	$t=8$	$t=9$	$t=10$	$t=11$	$t \in [12, \infty)$
N_2	8	8	8	8	14	14	14	20	20	20	24	24	24	24
N_3	8	8	8	8	14	14	14	20	20	20	22*	22	22	22
N_4	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

(d). Network delay table of N_4.

Table 2.2. Network delay tables of N_1, N_2, N_3 and N_4 under the 1-st order routing strategy where N_5 is the destination node.

network delay table of each node by keeping in network delay tables not only the delay of each path but also the set of nodes which follow the first node in the ordered sequence representation of the path. Thus, the format of routing messages is modified to include the corresponding set of nodes in each path. The augmented information in the network delay table of each node will be very useful in forming suitable routing messages for its neighbors. The number of nodes included in the routing message will henceforth be referred to as the *order* of the routing strategy. As the order of the routing strategy gets higher, it will become free of longer loops.

The main schemes used in all k-th order routing strategies are basically the same, except that different values of k indicate different amounts of information to be recorded in the network delay table. Let $NT^k_{i\hat{y}d}(m)$ denote the information kept in the network delay table of N_i about the shortest delay path from N_i via $N_j \in A_i$ to N_d under the k-th order routing strategy during the time interval [m, m+1). Also, let $P_{\hat{i}\hat{y},d}(m)$ be the path specified by $NT^k_{i\hat{y}d}(m)$. Then, $NT^k_{i\hat{y}d}(m)$ is a record containing two fields: $NT.dly^k_{\hat{i}\hat{y},d}(m)$ and $NT.set^k_{\hat{i}\hat{y},d}(m)$, where $NT.dly^k_{\hat{i}\hat{y},d}(m)$ denotes the delay of $P_{\hat{i}\hat{y},d}(m)$ and $NT.set^k_{\hat{i}\hat{y},d}(m)$ is an ordered set of the first k+1 nodes in $P_{\hat{i}\hat{y},d}(m)$. That is, $NT.dly^k_{\hat{i}\hat{y},d}(m) = d(P_{\hat{i}\hat{y},d})(m)$ and $NT.set^k_{\hat{i}\hat{y},d}(m) = H_{k+1}(P_{\hat{i}\hat{y},d}(m))$.

Let $RM^k_{i\leftarrow j,d}(m)$ denote the routing message sent from $N_j \in A_i$ to N_i about the optimal path from N_j to N_d under the k-th order routing strategy during the time interval [m, m+1). $RM^k_{i\leftarrow j,d}(m)$ is again composed of two fields, $RM.dly^k_{i\leftarrow j,d}(m)$ and $RM.set^k_{i\leftarrow j,d}(m)$, which can be determined from the network delay table of N_j as follows.

$$RM.dly^k_{i\leftarrow j,d}(m) = \min_{\substack{N_q \in A_j \text{ and} \\ N_i \notin NT.set^k_{\hat{j}q,d}(m)}} NT.dly^k_{\hat{j}q,d}(m), \tag{2.4}$$

$$RM.set^k_{i\leftarrow j,d}(m) = H_k(P_{i^*}) \quad \text{where } P_{i^*} \text{ is the path with the delay } RM.dly^k_{i\leftarrow j,d}(m). \tag{2.5}$$

When the routing message $RM^k_{i\leftarrow j,d}(m)$ is received by N_i, N_i uses this message to update its network delay table as follows, where $\odot$ means prefixing a node to an ordered set.

$$NT.dly^k_{Nj,d}(m) = RM.dly^k_{i\leftarrow j,d}(m-1) + DL_{i,j}(m), \tag{2.6}$$

$$NT.set^k_{Nj,d}(m) = \{N_i\} \odot RM.set^k_{i\leftarrow j,d}(m-1). \tag{2.7}$$

In addition, $DOP^k_{i,d}(m)$ represents the shortest delay from N_i to N_d in the network delay table of N_i under the k-th order routing strategy during the interval [m, m+1), i.e., $DOP^k_{i,d}(m) = \min_{N_j \in A_i} NT.dly^k_{Nj,d}(m)$, and denote the corresponding path as $OP^k_{i,d}(m)$. Notice that APRS and the routing strategy described Eq. (2.3) are actually special cases of the above strategy when k=0 and k=1, respectively. For the network in Figure 2.2, the network operations under the second order routing strategy are described in Table 2.3, where the subscript of each entry in the network delay tables represents the set of the second and third nodes of the corresponding path. If enough routing information is recorded, a node can determine that the use of some of its neighbors will not lead to loop-free paths; such neighbors will be removed from the construction of loop-free paths. The entries in Table 2.3 marked by ~ represent such cases. Also, in Table 2.3 the appearance of the first node in $NT.set^2_{Nj,d}$ that is N_i is omitted for simplicity. It is interesting to see that the second order routing strategy eliminates the first order loop (N_2, N_4, N_2) and the second order loop (N_2, N_4, N_3, N_2), which had caused the slowdown of the recovery processes in Tables 2.1 and 2.2, respectively. As a result, the required time intervals for N_1, N_2, N_3 and N_4 to get their new optimal paths to N_5 are reduced respectively to 6, 5, 5 and 4. It can be seen that increasing the order of routing strategy speeds up each node's adaptation to failures of links/nodes in the network.

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t \in [6, \infty)$
N_2	$7_{(2,4)}$	$7_{(2,4)}$	$7_{(2,4)}$	$11_{(2,3)}$	$19_{(2,4)}$	$19_{(2,4)}$	$19_{(2,4)}$	$27_{(2,4)}$
N_3	$9_{(3,4)}$	$9_{(3,4)}$	$9_{(3,4)}$	$11_{(3,2)}$	$21_{(3,4)}$	$21_{(3,4)}$	$21_{(3,4)}$	$25_{(3,5)}$'

(a). Network delay table of N_1.

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t \in [7, \infty)$
N_1	$13_{(1,3)}$	$13_{(1,3)}$	$13_{(1,3)}$	$13_{(1,3)}$	$\sim$	$25_{(1,3)}$	$25_{(1,3)}$	$25_{(1,3)}$	$29_{(1,3)}$
N_3	$7_{(3,4)}$	$7_{(3,4)}$	$7_{(3,4)}$	$23_{(3,5)}$	$23_{(3,5)}$	$23_{(3,5)}$	$23_{(3,5)}$*	$23_{(3,5)}$	$23_{(3,5)}$
N_4	$3_{(4,5)}$	$3_{(4,5)}$	$15_{(4,3)}$	$15_{(4,3)}$	$15_{(4,3)}$	$15_{(4,3)}$	$23_{(4,3)}$*	$23_{(4,3)}$	$23_{(4,3)}$

(b). Network delay table of N_2.

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t \in [7, \infty)$
N_1	$12_{(1,2)}$	$12_{(1,2)}$	$12_{(1,2)}$	$12_{(1,2)}$	$\sim$	$24_{(1,2)}$	$24_{(1,2)}$	$24_{(1,2)}$	$32_{(1,2)}$
N_2	$8_{(2,4)}$	$8_{(2,4)}$	$8_{(2,4)}$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
N_4	$4_{(4,5)}$	$4_{(4,5)}$	$16_{(4,2)}$	$16_{(4,2)}$	$16_{(4,2)}$	$16_{(4,2)}$	$\sim$	$\sim$	$\sim$
N_5	$20_{(5)}$	$20_{(5)}$	$20_{(5)}$	$20_{(5)}$	$20_{(5)}$	$20_{(5)}$	$20_{(5)}$*	$20_{(5)}$	$20_{(5)}$

(c). Network delay table of N_3.

entry	$t_0 \in (-\infty, 0)$	$t=0$	$t=1$	$t=2$	$t=3$	$t \in [4, \infty)$
N_2	$14_{(2,1)}$	$14_{(2,1)}$	$14_{(2,1)}$	$14_{(2,1)}$	$14_{(2,1)}$	$24_{(2,3)}$
N_3	$14_{(3,1)}$	$14_{(3,1)}$	$14_{(3,1)}$	$14_{(3,1)}$	$14_{(3,1)}$	$22_{(3,5)}$'
N_5	$2_{(5)}$	∞	∞	∞	∞	∞

(d). Network delay table of N_4.

Table 2.3. Network delay tables of N_1, N_2, N_3 and N_4 under the 2-nd order routing strategy where N_5 is the destination node.

2.1.3. Analysis of high order routing strategies

Under the k-th order routing strategy, each node gathers routing information and then updates its local delay table by exchanging routing messages with all its neighbors every fixed time interval. From Eqs. (2.4) and (2.5), one can see that under the k-th order routing strategy, each node, say N_i, by sending appropriate routing messages to neighboring nodes $N_q \in A_i$, will avoid forming any path containing a j-th order loop, $1 \le j \le k$, in the network delay table of N_q. More formally, we describe this fact by the lemma below.

Lemma 2.1: All paths implied by the network delay table of each node under the k-th order routing strategy are k-th order loop-free.

From the above lemma, it can be seen that in case of a link failure, a node under a higher order routing strategy, while being free of longer loops, can determine a new nonfaulty optimal path faster than under a lower order routing strategy, and the network's adaptability is thus enhanced. Since the information that the link $L_{f,f'}=L_{i_k,i_{k+1}}$ of a path has become faulty is broadcast only one hop per unit time under the high order routing strategies, it will take the same number of time units as the screen value of that path for the source node to be informed about the link failure. Suppose the link failure occurred at t=0, then the screen value of every path implied by the network delay table of each node under the k-th order routing strategy during the time interval [m, m+1) must be greater than m. This can be formally stated by the lemma below.

Lemma 2.2: Suppose the link failure occurred at time t=0, and let $P_{Nj,d}(m)$ be the path implied by $NT^k_{Nj,d}(m)$ in the network delay table of N_i. Then, $scn(P_{Nj,d}(m)) > m$.

Generally speaking, the screen value of a path means the number of time intervals required to remove this path from the network delay table of N_s after the occurrence of a failure in the path. Thus, under the k-th order routing strategy, the path with the shortest

delay to a destination node N_d chosen by a source node N_s is the shortest among all possible paths which have not been found faulty by N_s up to time m. More formally, we have the following theorem.

Theorem 2.1:

$$DOP^k_{s,d}(m) = \min\left\{ d(P_j) \,\middle|\, P_j \in SP^k_{s,d} \text{ and } scn(P_j) > m \right\} \quad \forall\, m \in I^+.$$

Proof: To facilitate our presentation, consider the APRS. Let m_0 be the time intervals required to find a new nonfaulty optimal path under the APRS in case of a link failure. Consider two cases: $m \geq m_0$ and $m < m_0$. When $m \geq m_0$, $OP_{s,d}(m)$ is the newly found optimal path and fault-free. That is, $scn(OP_{s,d}(m)) = \infty$, resulting in

$$DOP_{s,d}(m) = \min\left\{ d(P_l) \,\middle|\, P_l \in SP_{s,d} \text{ and } scn(P_l) = \infty \right\}.$$

When $m < m_0$, $OP_{s,d}(m)$ still contains the faulty link L_{f,f^*}. We want to show that $OP_{s,d}(m)$ chosen at time m from the local delay table of N_s is indeed the real minimum delay path found by N_s. Let P_{l^*} be an arbitrary path containing the faulty link L_{f,f^*} at time m, $P_{l^*} \in SP_{s,d}$ and $scn(P_{l^*}) > m$. Then, P_{l^*} can be expressed as $(N_{i_0^*}, N_{i_1^*}, \cdots, N_{i_r^*}, N_{i_{r+1}^*}, \cdots, N_{i_u^*})$, where $N_{i_0^*} = N_s$, $N_{i_r^*} = N_f$, $N_{i_{r+1}^*} = N_{f^*}$, $N_{i_u^*} = N_d$, and $scn(P_{l^*}) = r > m$. From Eqs. (2.1) and (2.2), we get

$$DOP^*_{i_0^*,i_u^*}(m) \leq DL_{i_0^*,i_1^*}(m) + DOP^*_{i_1^*,i_u^*}(m-1)$$

$$DOP^*_{i_1^*,i_u^*}(m-1) \leq DL_{i_1^*,i_2^*}(m-1) + DOP^*_{i_2^*,i_u^*}(m-2)$$

$$\vdots$$

$$DOP^*_{i_{r-1}^*,i_u^*}(m-r+1) \leq DL_{i_{r-1}^*,i_r^*}(m-r+1) + DOP^*_{i_r^*,i_u^*}(m-r)$$

$$\Rightarrow \quad DOP_{i_0^*,i_u^*}(m) \le \sum_{k=0}^{r-1} DL_{i_k^*,i_{k+1}^*}(m-k) + DOP_{i_r^*,i_u^*}(m-r).$$

Since r>m, m−r is a time before $L_{i_r^*,i_{r+1}^*}$ became faulty and we get

$$DOP_{i_r^*,i_u^*}(m-r) = \sum_{k=r}^{u-1} DL_{i_k^*,i_{k+1}^*}(m-k)$$

$$\Rightarrow \quad DOP_{s,d}(m) = DOP_{i_0^*,i_u^*}(m) \le \sum_{k=0}^{u-1} DL_{i_k^*,i_{k+1}^*}(m-k) = d(P_i^*).$$

From Lemma 2.2, $scn(OP_{s,d}(m)) > m$ and $OP_{s,d}(m) \in SP_{s,d}$, thereby proving this theorem for the APRS (the 0-th order routing strategy).

For the higher order routing strategies, it can be proved similarly by recursively applying Eqs. (2.4) and (2.6) and retrospectively tracing the network delay tables of nodes in an arbitrary path. **Q.E.D.**

It can be verified by Theorem 2.1 that the delays of nonfaulty optimal paths from N_s to N_d obtained under all strategies are indeed the same, since the new optimal path must be free of any loop, i.e., $DOP^i_{s,d}(m_i) = DOP^j_{s,d}(m_j) = r$, $\forall$ i, j $\in \mathbf{I}^+$, where m_k is the number of time intervals for N_s to determine a new nonfaulty optimal path to N_d under the k-th order routing strategy after the occurrence of a link failure. Also, it can be seen that $DOP^k_{s,d}(m) \le DOP^k_{s,d}(m+1)$ $\forall$ m $\in$ I, meaning that after a link failure the delay of the shortest path chosen by the k-th order routing strategy is a nondecreasing function of time. Moreover, we have the following theorem to determine m_k $\forall$ k $\in \mathbf{I}^+$, where $r < \infty$ denotes the new nonfaulty optimal path.

Theorem 2.2: $$m_k = \max\left\{ scn(P_j) \mid P_j \in SP^k_{s,d} \text{ and } d(P_j) < r \right\}.$$

Proof: Suppose $m_k^* = \max\left\{ scn(P_i) \mid P_i \in SP^k_{s,d} \text{ and } d(P_i) < r \right\}$ and $m_k^* \neq m_k$. We shall

prove that both the assumptions of $m_k > m_k^*$ and $m_k < m_k^*$ lead to contradictions.

Case 1: $m_k > m_k^*$.

Since m_k is the minimal number of time intervals required for N_s to obtain the new nonfaulty optimal path to N_d, $DOP^k_{s,d}(m_k-1) < DOP^k_{s,d}(m_k) = r$. By Theorem 2.1, $scn(OP^k_{s,d}(m_k-1)) = m_k^{**} > m_k - 1 \Rightarrow m_k^{**} \geq m_k$. Because $SP^k_{s,d}$ is the set of all k-th order loop-free paths from N_s to N_d, and $OP^k_{s,d}(m_k-1) \in SP^k_{s,d}$ from Lemma 2.1,

$$m_k^* = \max\left\{ scn(P_i) \mid P_i \in SP^k_{s,d} \text{ and } d(P_i) < r \right\} \geq m_k^{**} \geq m_k,$$

leading to a contradiction to $m_k^* < m_k$.

Case 2: $m_k < m_k^*$.

Since $m_k^* = \max\left\{ scn(P_i) \mid P_i \in SP^k_{s,d} \text{ and } d(P_i) < r \right\}$, there exists a path P_j such that $scn(P_j) = m_k^*$, $P_j \in SP^k_{s,d}$ and $d(P_j) < r$. From Theorem 2.1, $DOP^k_{s,d}(m_k^*-1) \leq d(P_j) < r$. For $m_k, m_k^* \in I^+$, $m_k < m_k^* \Rightarrow m_k \leq m_k^* - 1$. Thus, we get $DOP^k_{s,d}(m_k) \leq DOP^k_{s,d}(m_k^*-1) \leq d(P_j) < r$. However, $r = DOP^k_{s,d}(m_k)$, leading to a contradiction. **Q.E.D.**

From this theorem, we get the following corollary, which implies that the network's adaptability is improved monotonically by adding more information in routing messages.

Corollary 2.2.1: $m_{k+1} \leq m_k \; \forall \; k \in I^+$.

By definition, we can describe the screen and delay functions of a path $P_i = (N_{i_0}, N_{i_1}, \ldots, N_{i_k}, N_{i_{k+1}}, \ldots, N_{i_u}) \in SP^k_{s,d}$ where $N_{i_0}=N_s$, $N_{i_k}=N_f$, $N_{i_{k+1}}=N_{f'}$, and $N_{i_u}=N_d$ as below.

$$scn(P_i) = h(P_i') + \sum_{j=1}^{k} n_{i_j} h(L_{i_j}), \tag{2.8}$$

$$d(P_i) = d(P_i') + \sum_{j=1}^{k} n_{i_j} d(L_{i_j}) + d(P_{i_r}), \tag{2.9}$$

where $P_i \supset P_{i_r} \in SP^k_{f,d}$, $P_i' \in SP^k_{s,f}$, L_{i_j} is a loop starting and ending at N_{i_j} and n_{i_j} is the number of times the loop L_{i_j} appears in the path P_i.

Since we can include all redundant parts of P_i in the second term of (2.8) and the second and third terms of (2.9), P_i' can be regarded as a path without loops. For the example network of Figure 2.3, consider a path $P_i = (N_1, N_2, N_3, N_4, N_2, N_3, N_5, N_3, N_5, N_3, N_5, N_6, N_7)$ in which the link $L_{5,6}$ is faulty. Then $P_i' = (N_1, N_2, N_3, N_5)$, $P_{i_r} = (N_5, N_6, N_7)$, $L_{i_1} = (N_2, N_3, N_4, N_2)$, $n_{i_1} = 1$ and $L_{i_2} = (N_3, N_5, N_3)$, $n_{i_2} = 2$. Then, in light of Eqs. (2.8), (2.9) and Theorem 2.2, m_k can be expressed as follows.

$$m_k : \max_{P_i \in SP^k_{f,d}} h(P_i') + \sum_{j=1}^{k} n_{i_j} h(L_{i_j})$$

$$\text{subject to } d(P_i') + \sum_{j=1}^{k} n_{i_j} d(L_{i_j}) + d(P_{i_r}) < r \quad \text{where } L_{i_j} \in SP^k_{i_j,i_j} \text{ and } L_{i_j} \subset P_i, \tag{2.10}$$

where P_i', P_{i_r}, n_{i_j} and L_{i_j} are defined as in Eqs. (2.8) and (2.9).

Define L_{k^*} to be a loop such that $ave(L_{k^*}) = \min\left\{ave(L_i) \mid L_i \in SP^k_{i,i},\ 1 \le i \le n\right\}$. When the delay of the new nonfaulty optimal path, r, is large enough, $ave(L_{k^*})$ will become the dominating factor in determining m_k and we have the following theorem.

Theorem 2.3: $\lim_{r\to\infty} \dfrac{m_i}{m_j} = \dfrac{ave(L_{i^*})}{ave(L_{j^*})}, \forall\ i, j \in \Gamma^+$.

Proof: Let L_{i_m} be the loop with the minimal average delay among all loops in the path P_i. From Eq. (2.10), it can be seen that L_{i_m} will be traveled as many rounds as possible in order to maximize m_k. Thus, we have

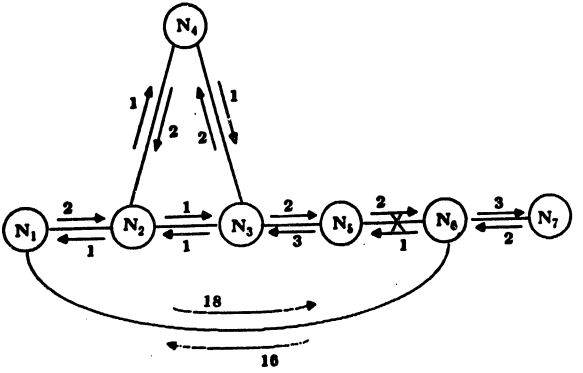


Figure 2.3. An example network for Eqs. (2.8) and (2.9).

$$m_k = \max_{P_i \in SP^k_{s,d}} \left\{ \left\lfloor \frac{r - d(P_i') - d(P_i)}{d(L_{i_m})} \right\rfloor h(L_{i_m}) + R_i + h(P_i') \right\} \quad \text{where} \quad R_i < h(L_{i_m}). \tag{2.11}$$

Clearly, the second and third terms in Eq. (2.11) become negligible as $r \to \infty$ and this theorem follows. **Q.E.D.**

The following example is presented to illustrate our analytic results obtained thus far.

Example 2.1: In a computer network shown in Figure 2.1, assume that the link $L_{3,4}$ became faulty at time 0 and delays of the other links remain constant. From Eq. (2.11), the number of time intervals required for each node in this network to obtain its new nonfaulty optimal path is a function of $r^* = L_{2,5}$. Thus, network performance has been examined while this parameter is being varied.

(I) Case 1: $r^* = 16$.

Given below are the operations taken under the APRS (the 0-th order routing strategy) and the 1-st order routing strategy for the source node N_1 to obtain a new nonfaulty optimal path to the destination node N_5 by exchanging routing messages with neighboring nodes every time interval.

Under the APRS, network delay tables become as shown in Table 2.4. From Table 2.4, one can obtain:

$OP_{1,5}(t_0) = (N_1,N_3,N_4,N_5)$, where $t_0 \in (-\infty, 0)$

$OP_{1,5}(0) = (N_1,N_3,N_4,N_5)$

$OP_{1,5}(1) = OP_{1,5}(2) = (N_1,N_3,N_1,N_3,N_4,N_5)$

$OP_{1,5}(3) = OP_{1,5}(4) = (N_1,N_3,N_1,N_3,N_1,N_3,N_4,N_5)$

$OP_{1,5}(5) = OP_{1,5}(6) = (N_1,N_3,N_1,N_3,N_1,N_3,N_1,N_3,N_4,N_5)$

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10	$t \in [11, \infty)$
N_2	11	11	11	11	14	14	17	17	19	19	19∘	19	19
N_3	6	6	9	9	12	12	15	15	18	18	21	21	22

(a). Network delay table of N_1.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10	$t \in [11, \infty)$
N_1	8	8	8	11	11	14	14	17	17	20	20	21	21
N_3	9	9	12	12	15	15	18	18	21	21	24	24	25
N_4	16	16	16	16	16	16	16	16∘	16	16	16	16	16

(b). Network delay table of N_2.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	$t \in [10, \infty)$
N_1	7	7	7	10	10	13	13	16	16	19	19	20∘
N_2	14	14	14	14	17	17	20	20	22	22	22	22
N_4	4	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

(c). Network delay table of N_3.

Table 2.4. Illustrative network delay tables under APRS for Example 2.1.

$OP_{1,5}(7) = OP_{1,5}(8) = (N_1,N_3,N_1,N_3,N_1,N_3,N_1,N_3,N_1,N_3,N_4,N_5)$

$OP_{1,5}(9) = (N_1,N_2,N_5)$.

Thus, $m_0 = 9$, while $scn(OP_{1,5}(8)) = 9$. This agrees with the result of Theorem 2.2. In addition, one can observe from the above tables that the numbers of time intervals for N_2 and N_3 to obtain the new nonfaulty optimal paths are 6 and 10, respectively (marked by *).

On the other hand, we obtain the network delay tables under the 1-st order routing strategy as shown in Table 2.5. From this table we get:

$OP^1_{1,5}(t_0) = (N_1,N_3,N_4,N_5)$, where $t_0 \in (-\infty, 0)$

$OP^1_{1,5}(0) = (N_1,N_3,N_4,N_5)$

$OP^1_{1,5}(1) = (N_1,N_2,N_3,N_4,N_5)$

$OP^1_{1,5}(2) = OP^1_{1,5}(3) = (N_1,N_3,N_2,N_1,N_3,N_4,N_5)$

$OP^1_{1,5}(4) = (N_1,N_2,N_5)$

Again, $m_1 = 4 = scn(OP^k_{1,5}(3))$. The numbers of time intervals required for N_2 and N_3 to obtain the new optimal paths are also reduced from 6 and 10 to 2 and 3, respectively.

(II) Case 2: $r^* = 160$.

Applying the same procedure as above, we can obtain the results for $r^* = 160$ with a simple calculation. The numbers of time intervals required for N_1, N_2 and N_3 to obtain their new optimal paths to N_5 are reduced from 105, 102 and 106 (under the APRS) to 50, 49 and 51 (under the 1-st order routing strategy), respectively.

(III) Case 3: $r^* = 1000$.

In this case, the numbers of time intervals for N_1, N_2 and N_3 to obtain their optimal paths to N_5 are 665, 662 and 666 under the APRS and 332, 331 and 333 under the 1-st order routing strategy.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	$t \in [4, \infty)$
N_2	12	12	12	19	19	19*
N_3	6	6	16	16	16	24

(a). Network delay table of N_1.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	t=3	t=4	$t \in [5, \infty)$
N_1	8	8	8	18	18	18	26
N_3	9	9	18	18	18	25	25
N_5	16	16	16	16*	16	16	16

(b). Network delay table of N_2.

entry	$t_0 \in (-\infty, 0)$	t=0	t=1	t=2	$t \in [3, \infty)$
N_1	13	13	13	13	20*
N_2	14	14	14	14	22
N_4	4	∞	∞	∞	∞

(c). Network delay table of N_3.

Table 2.5. Illustrative network delay tables under the 1-st order routing strategy for Example 2.1.

In Figure 2.1, the possible loop with the minimal average delay under the APRS is L_A = (N_1, N_3, N_1), and that under the 1-st order routing strategy is L_B = (N_1, N_2, N_3, N_1). Since $d(L_A) = 3$, $h(L_A) = 2$, $d(L_B) = 9$ and $h(L_B) = 3$, $ave(L_A) = 1.5$ and $ave(L_B) = 3$. The results of Cases 2 and 3 agree with Theorem 2.3.

The above results indicate that the network's adaptability is actually enhanced with the increased amount of local information. However, multi-node loops may occur very infrequently and the operational overheads required in higher-order routing strategies cannot be ignored. Thus, the feasibility of implementing higher-order routing strategies may need further justification. Notice that although a higher order routing strategy is necessary for some nodes to avoid potential looping, it may contribute nothing but higher operational overheads to other nodes. Consequently, it is very important for us to determine the minimal order routing strategy required for *each* node to remove looping from its network delay table.

2.2. Minimal Order Loop-Free Routing Strategy

In Section 2.2.1, we shall formulate the method to determine the minimal order routing strategy required for each node to avoid all potential looping. The tradeoff between the operational overhead and looping delay will be investigated and determined in Section 2.2.2. The complexity of the optimization procedure will also be examined.

2.2.1. Determination of minimal order strategy for loop-free

Consider the case when $L_{i,j}$ becomes faulty. Let $R_{i\leftarrow k,j}$ denote the required order of routing message sent from $N_k \in A_i$ to N_i such that the routing message will not result in any path containing loops in the network delay table of N_i. To facilitate our presentation, we define a set of loops $S_{i\leftarrow k,j}$ as follows.

$$S_{i\leftarrow k,j} \equiv \left\{ L_{i^*} \mid L_{i^*} \in SL_{i,i},\ 2nd(L_{i^*})=N_k \text{ and } d(L_{i^*}) < d(P_{i,j-i,j}) - DL_{i,j} \right\}, \qquad (2.12)$$

where $N_i \in V(N)$, $N_k, N_j \in A_i$, and $2nd(L_{i^*})$ is the second node in the loop L_{i^*}. Then, the quantity $R_{i\leftarrow k,j}$ can be determined by the following theorem.

Theorem 2.4:

$$R_{i\leftarrow k,j} = \begin{cases} \max\limits_{L_{i^*} \in S_{i\leftarrow k,j}} \left\{ h(L_{i^*}) \right\}, & \text{if } S_{i\leftarrow k,j} \neq \varnothing, \\ 0, & \text{if } S_{i\leftarrow k,j} = \varnothing. \end{cases}$$

Proof: If the required order of the routing message the destination node N_j from $N_k \in A_i$ to N_i, denoted by $r_{i\leftarrow k,j}$, is less than $R_{i\leftarrow k,j}$, there is a loop $L_{i^*} \in SL_{i,i}$ such that $2nd(L_{i^*}) = N_k$ and $d(L_{i^*}) + DL_{i,j} < d(P_{i,j-i,j})$. Thus, the path from N_k via L_{i^*} to N_i and then via $L_{i,j}$ to N_j contains the delay $d(L_{i^*}) - DL_{i,k} + DL_{i,j}$. The delay of the new optimal path from N_k to N_j must be greater than $d(L_{i^*}) - DL_{i,k} + DL_{i,j}$, since $d(P_{i,j-i,j})$ will otherwise be less than $d(L_{i^*}) + DL_{i,j}$, leading to a contradiction. Thus, if $r_{i\leftarrow k,j} < h(L_{i^*})$ before N_k finds its new optimal path, the delay $d(L_{i^*}) - DL_{i,k} + DL_{i,j}$ will be sent to N_i, thereby resulting in a path with a loop in the network delay table of N_i. Therefore, $R_{i\leftarrow k,j} \leq r_{i\leftarrow k,j}$.

Next, we want to prove that a routing message of the order $R_{i\leftarrow k,j}$ sent from N_k to N_i will not result in a path with loops for N_i. Suppose there is a resulting loop L_{i^*}. Then, $d(L_{i^*}) + DL_{i,j} < d(P_{i,j-i,j})$ and $2nd(L_{i^*})=N_k$. By Eqs. (2.5) and (2.7), we get $h(L_{i^*}) > R_{i\leftarrow k,j}$, leading to a contradiction. This means $R_{i\leftarrow k,j} \geq r_{i\leftarrow k,j}$, and $R_{i\leftarrow k,j} = r_{i\leftarrow k,j}$ thus follows. **Q.E.D.**

Note that the minimal order routing strategy for N_i must be determined by routing messages from all of its neighboring nodes. Let $R_{i\leftarrow k}$ represent the order of routing message sent from $N_k \in A_i$ to N_i to avoid all looping in the network delay table of N_i, i.e.,

$$R_{i\leftarrow k} = \max_{N_j \in A_i \text{ and } j\neq k} \left\{ R_{i\leftarrow k,j} \right\}. \qquad (2.13)$$

The minimal order of routing strategy required for N_i, denoted by O_i^*, can be determined by the following corollary.

Corollary 2.4.1: All paths implied by the network delay table of each node are free of looping if and only if

$$O_i^* = \max_{N_j \in A_i} \left\{ R_{j\leftarrow i} \right\}, \quad \forall\, N_i \in V(N).$$

Proof: If $O_i^* = \max_{N_j \in A_i} \left\{ R_{j\leftarrow i} \right\}$, then it immediately follows from Theorem 2.4 that the routing message sent from N_i will not result in any path with loops in the network delay table of a receiving node when N_i adopts the O_i^*-th order routing strategy. Next, we want to prove that O_i^* is the minimal order of routing strategy for N_i to avoid resulting any path with loops in the network delay tables of those nodes receiving routing messages from N_i. Suppose that the order of routing strategy adopted by N_i, denoted by O_i, is less than O_i^*. Then there exists an $R_{j\leftarrow i}$ such that $R_{j\leftarrow i} > O_i$. From the equations in Theorem 2.4, there is a node N_k such that $R_{j\leftarrow i,k} > O_i$. Thus, when the link $L_{j,k}$ becomes faulty, the routing message sent from N_i to N_j will result in a path with an $R_{j\leftarrow i}$-th order loop in the network delay table of N_j, where $O_i < R_{j\leftarrow i} \leq O_i^*$, leading to a contradiction. **Q.E.D.**

Although the above formulas can determine the minimal order routing strategy for each node, one can find from the operation of routing strategy that the difference between the orders of routing strategies of two adjacent nodes cannot be greater than one. (We term this fact the "strategy compatibility".) Otherwise, a node with the lower order routing strategy would not be able to generate the routing messages required for all of its neighboring nodes. Thus,

we may have to increase the orders of routing strategies of some nodes to hold the strategy compatibility. We present a simple example to demonstrate the ideas presented thus far.

Example 2.2: Consider the example network in Figure 2.4. For this network we will determine the minimal order of loop-free routing strategy for each node.

(a). The required order of loop-free routing strategy, O_i^*, $1 \le i \le 8$, can be determined by the following two steps.

Step 1. Using Theorem 2.4 and Eq. (2.13), determine $R_{i\leftarrow j}$, $N_j \in A_i$, $1 \le i \le 8$. For N_1, we get

$$\begin{cases} R_{1\leftarrow 4,2} = 0 \\ R_{1\leftarrow 4,3} = 0 \end{cases} \Rightarrow R_{1\leftarrow 4} = 0$$

$$\begin{cases} R_{1\leftarrow 3,2} = 0 \\ R_{1\leftarrow 3,4} = 0 \end{cases} \Rightarrow R_{1\leftarrow 3} = 0$$

$$\begin{cases} R_{1\leftarrow 2,3} = 0 \\ R_{1\leftarrow 2,4} = 1 \end{cases} \Rightarrow R_{1\leftarrow 2} = 1.$$

For N_2, we have $R_{2\leftarrow 1,3} = R_{2\leftarrow 1} = 1$ and $R_{2\leftarrow 3,1} = R_{2\leftarrow 3} = 1$.

In case of N_3, we obtain

$$\begin{cases} R_{3\leftarrow 1,2} = 0 \\ R_{3\leftarrow 1,4} = 0 \\ R_{3\leftarrow 1,8} = 0 \end{cases} \Rightarrow R_{3\leftarrow 1} = 0$$

$$\begin{cases} R_{3\leftarrow 2,1} = 0 \\ R_{3\leftarrow 2,4} = 1 \\ R_{3\leftarrow 2,8} = 2 \end{cases} \Rightarrow R_{3\leftarrow 2} = 2$$

and then, $R_{3\leftarrow 4,1} = 0$, $R_{3\leftarrow 4,2} = 1$, $R_{3\leftarrow 4,8} = 1 \Rightarrow R_{3\leftarrow 4} = 2$, and $R_{3\leftarrow 8,1} = 0$, $R_{3\leftarrow 8,2} = 0$, $R_{3\leftarrow 8,4} = 0 \Rightarrow R_{3\leftarrow 8} = 0$. Following the same procedure, we get $R_{4\leftarrow 1} = 3$, $R_{4\leftarrow 3} = 3$ and $R_{4\leftarrow 5} = 0$ for N_4, $R_{5\leftarrow 6} = 1$ and $R_{5\leftarrow 4} = 1$ for N_5, $R_{6\leftarrow 5} = 1$ and $R_{6\leftarrow 7} = 1$ for N_6, $R_{7\leftarrow 6} = 1$

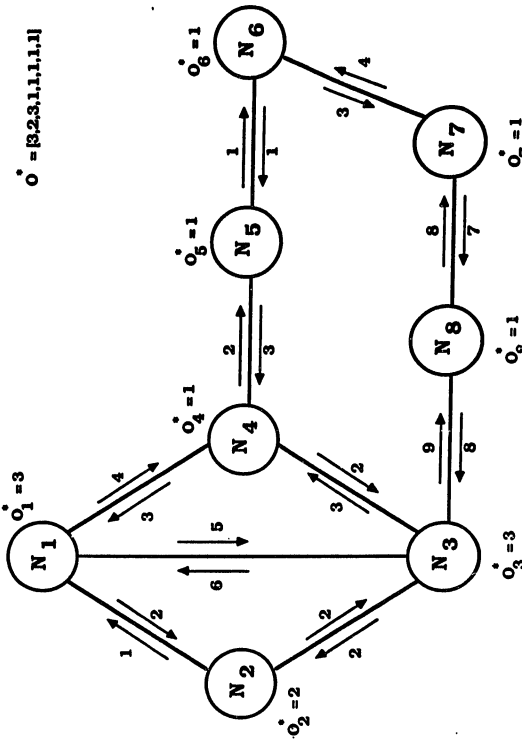


Figure 2.4. A network for Example 2.2.

and $R_{7\leftarrow 8} = 1$ for N_7, and $R_{8\leftarrow 7} = 0$ and $R_{8\leftarrow 3} = 0$ for N_8.

Step 2. Using Corollary 2.4.1, determine O_i^*, $1 \le i \le 8$.

$$\begin{cases} R_{2\leftarrow 1} = 1 \\ R_{3\leftarrow 1} = 0 \\ R_{4\leftarrow 1} = 3 \end{cases} \Rightarrow O_1^* = 3$$

$$\begin{cases} R_{1\leftarrow 2} = 1 \\ R_{3\leftarrow 2} = 2 \end{cases} \Rightarrow O_2^* = 2$$

$$\begin{cases} R_{2\leftarrow 3} = 1 \\ R_{1\leftarrow 3} = 0 \\ R_{4\leftarrow 3} = 3 \\ R_{8\leftarrow 3} = 0 \end{cases} \Rightarrow O_3^* = 3$$

$$\begin{cases} R_{1\leftarrow 4} = 0 \\ R_{3\leftarrow 4} = 1 \\ R_{5\leftarrow 4} = 1 \end{cases} \Rightarrow O_4^* = 1$$

and then, $R_{4\leftarrow 5} = 0$, $R_{6\leftarrow 5} = 1 \Rightarrow O_5^* = 1$; $R_{5\leftarrow 6} = 1$, $R_{7\leftarrow 6} = 1 \Rightarrow O_6^* = 1$; $R_{6\leftarrow 7} = 1$, $R_{8\leftarrow 7} = 0 \Rightarrow O_7^* = 1$ and $R_{3\leftarrow 8} = 0$, $R_{7\leftarrow 8} = 1 \Rightarrow O_8^* = 1$. For this example network we get the minimum order vector, $O^* = [3, 2, 3, 1, 1, 1, 1, 1]$, and then [3, 2, 3, 2, 1, 1, 1, 2] after considering the strategy compatibility.

2.2.2. Operational overhead and looping delay tradeoff

As mentioned earlier, the multi-order routing strategy in a node usually causes its neighboring nodes to increase their orders of routing strategies to satisfy the strategy compatibility. If we consider the operational overhead in handling routing messages, it may not be worth introducing a considerable amount of overhead for infrequent failures or for some failures whose recovery costs are not high. This implies the need of striking a compromise between looping effects and the operational overhead, and determining the optimal order of routing

strategy for each node.

2.2.2.1. optimization of tradeoff

Although various procedures are conceivable to determine the operational overhead in Eqs. (2.4) and (2.5), the main idea can be described as follows. The cardinality of $RM.set^{k}_{i\leftarrow j,d}$ increases linearly with the order of routing strategy, meaning that the memory requirement for the routing strategy is linearly dependent on the value of k. The computational overhead for Eqs. (2.6) and (2.7) is straightforward and has little dependence on k. However, from Eq. (2.4) the number of comparisons required is linearly proportional to k. The computational cost is therefore linearly proportional to k. Following the above reasoning, the cost required for a node adopting the k-th order strategy to generate and process a routing message, denoted by $R_c(k)$, can be approximately expressed as a*k + *offset*, where *offset* is the sum of contributions from the factors unrelated to k.

Define a *strategy vector* as a p-tuple whose i-th element is the order of the routing strategy adopted by N_i, where p is the number of nodes in the network. A network together with its adopted strategy vector is termed a *configuration*. Let O_i^k denote the order of the routing strategy adopted by N_i when the configuration is C_k. The operational overhead per second induced with the configuration C_k can then be determined by the formula:

$$RC(C_k) = \sum_{i=1}^{p} R_c(O_i^k). \tag{2.14}$$

Assume that the traffic density between every pair of nodes in the network is uniform. The expected number of time intervals required for an arbitrary node to find a new nonfaulty optimal path to any other node when $L_{i,j}$ becomes faulty can be expressed as:

$$R_t(L_{i,j};\, C_k) = \frac{1}{p(p-1)} \sum_{N_u \in V(N)} \sum_{\substack{N_v \in V(N) \\ \text{and } u \neq v}} m_{u,v-i,j}(C_k), \tag{2.15}$$

where $m_{u,v-i,j}(C_k)$ denotes the number of time intervals for N_u to obtain a new nonfaulty optimal path to N_v when the configuration is C_k and $L_{i,j}$ becomes faulty. The expected number of time intervals to recover from an arbitrary link failure (i.e., switch from a broken path to a new nonfaulty path) in the configuration C_k can then be determined by:

$$RT(C_k) = \frac{1}{q} \sum_{L_{i,j} \in E(N)} R_t(L_{i,j};\, C_k). \tag{2.16}$$

Note that $RT(C_k)$ can be viewed as a *measure of adaptability* of C_k. The smaller $RT(C_k)$, the better adaptability C_k possesses. To compute Eq. (2.15), we must show how to determine $m_{u,v-i,j}(C_k)$ $\forall$ u, v, i, j, and k. Consider the case when in a configuration C_k, N_i does not adopt a routing strategy of an order sufficient enough to remove looping completely. In such a case, by Corollary 2.4.1, certain link failures will induce looping. From Eq. (2.13) and Theorem 2.4, we can represent the set of loops (SPL) induced by the insufficient order of routing strategy as follows.

$$SPL(N_i;\, C_k) = \bigcup_{N_j \in A_i} \bigcup_{\substack{N_q \in A_j \\ \text{and } q \neq i}} \left\{ L_{j^*} \mid L_{j^*} \in S_{j \leftarrow i,q} \text{ and } h(L_{j^*}) > O_i^k \right\}. \tag{2.17}$$

The set of all potential loops in the network with the configuration C_k can be expressed by:

$$SPL(C_k) = \bigcup_{N_i \in V(N)} SPL(N_i;\, C_k). \tag{2.18}$$

Let $L(P_{i^*})$ denote the set of loops in the path P_{i^*}. P_{i^*} is said to be a *possible path* in the configuration C_k if $L(P_{i^*}) \subseteq SPL(C_k)$, i.e., every loop contained in P_{i^*} belongs to $SPL(C_k)$.

Let Sb denote a subset of SPL(C_k) and V(Sb) be the set of the starting nodes of all the loops in Sb. Also, let $SP_{u\backslash V(Sb),\,i}$ be the set of paths from N_u to N_i which go through each node in V(Sb) at least once. Then, $m_{u,v-i,j}(C_k)$ can be formulated as follows.

$$m_{u,v-i,j}(C_k) = \max_{Sb \subseteq SPL(C_k)} \max_{\substack{L(P_{u^*}) \subseteq Sb \text{ and} \\ P_{u^*} \in SP_{u\backslash V(Sb),\,i}}} \left\{ h(P_{u^*}) \mid d(P_{u^*}) < d(P_{u,v-i,j}) - d(P_{i,v}) \right\}. \tag{2.19}$$

Once the network is given, using the above algorithm we can obtain $m_{u,v-i,j}(C_k)$ for all N_u, $N_v \in V(N)$ and $L_{i,j} \in E(N)$, and then RT(C_k) from Eqs. (2.15) and (2.16). RC(C_k) is determined by Eq. (2.14). Since the required order of routing strategy for each node can be obtained by Corollary 2.4.1, the number of possible configurations under the constraint of strategy compatibility can thus be determined. Once a design objective function F(C_k) = f(RC(C_k), RT(C_k)) is decided, the optimal configuration can be determined by evaluating each possible configuration.

Note that instead of exhaustively evaluating all possible strategy vectors, we can skip the evaluation of the configurations in either of the following two cases: (i) there is a node assigned a routing strategy of an order higher than it requires, i.e., OV[i] > O_i^* and OV[i] ≥ OV[j] $\forall$ $N_j \in A_i$, where OV[i] denotes the order of routing strategy for N_i $\forall$ $N_i \in V(N)$, and (ii) the difference in the order of strategy between any two adjacent nodes is greater than one. Clearly, the knowledge of the minimal order for loop-free routing and the strategy compatibility reduces the number of configurations to be evaluated significantly. Configurations of the example network in Figure 2.4 are evaluated in the following sequence.

[3, 2, 3, 2, 1, 1, 1, 2] (evaluated)

[3, 2, 3, 2, 1, 1, 1, 1] (OV[3] − OV[8] > 1 ⇒ skipped)

[3, 2, 3, 2, 1, 1, 0, 2] (OV[8] − OV[7] > 1 ⇒ skipped)

[3, 2, 3, 2, 1, 1, 0, 1] (OV[3] – OV[8] > 1 ⇏ skipped)

[3, 2, 3, 2, 1, 0, 1, 2] (evaluated)

⋮

[3, 2, 2, 2, 1, 1, 1, 2] (OV[8] > O_8^* and (OV[8] ≥ OV[3], OV[7] ⇏ skipped)

[3, 2, 2, 2, 1, 1, 1, 1] (evaluated)

⋮

[0, 0, 0, 0, 0, 0, 0, 0] (evaluated)

2.2.2.2. the number of configurations to be evaluated

For each configuration C_k, the number of $m_{u,v-i,j}(C_k)$'s needed to obtain $F(C_k)$ is p(p−1)q, where p=|V(*N*)| and q=|E(*N*)|. That is, the operation in Eq. (2.19) has to be executed p(p−1)q times for each configuration. Therefore, the number of configurations to be evaluated is a dominating factor to the execution time of the whole procedure.

To estimate the number of configurations to be evaluated, consider the following interesting combinatorial problem first. Given a labeled graph, if we want to assign each node with an integer chosen from $\mathbf{I}_m \equiv \{0,1,2, \cdots ,m\}$ in such a way that the difference between the numbers assigned to any two adjacent nodes must be less than or equal to one, how many assignments are there? Notice that if the labeled graph is G=(V(*N*), E(*N*)), then the answer to the above problem is exactly the number of possible configurations in the case of $O_i^* = m$ $\forall$ $N_i \in$ V(*N*).

Define a *distribution vector* (*D-vector*) D_i for each node N_i, the k-th component of which, denoted by $D_i(k)$, represents the number of times N_i is assigned the value $k \in \mathbf{I}_m$. Figure 2.5 is an illustration of this idea with m = 3. Consider the case when one more node N_g

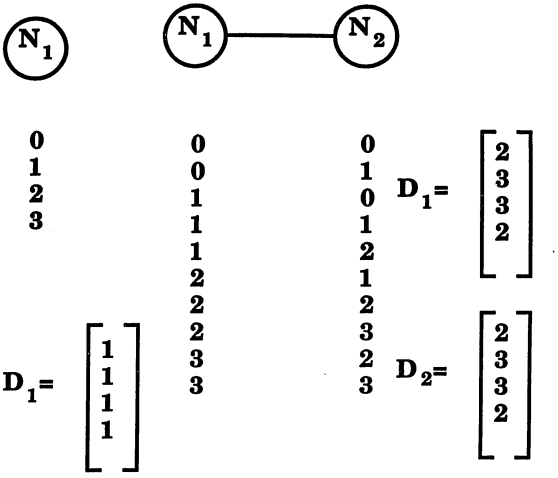


Figure 2.5. Illustration of D-vectors.

is to be attached to a node N_d in a given graph. The D-vectors of N_g and N_d in the resulting graph are denoted by D_g and D_d, respectively, while the D-vector of N_d in the original graph is represented by D_d'. Then, the relationship between these quantities can be determined by the following lemma.

Lemma 2.3:

$$\text{(a)} \begin{cases} D_g(0) = D_d'(0) + D_d'(1). \\ D_g(i) = \sum_{j=i-1}^{j=i+1} D_d'(j), \quad 1 \le i \le m-1 \\ D_g(m) = D_d'(m-1) + D_d'(m). \end{cases}$$

$$\text{(b)} \begin{cases} D_d(0) = D_d'(1)*2. \\ D_d(i) = D_d'(i)*3, \quad 1 \le i \le m-1 \\ D_d(m) = D_d'(m)*2. \end{cases}$$

Proof: (a) Suppose the node N_g is assigned 0. Then, it can be attached to N_d only when N_d was originally assigned 0 or 1. Thus, $D_g(0) = D_d'(0) + D_d'(1)$. Similarly, we can get the other two equations.

(b) When N_d is assigned 0, possible numbers assigned to N_g are 0 and 1, each of which corresponds to a different assignment in the resulting graph. Thus, $D_d(0) = D_d'(0)*2$. The other two equations in (b) can be obtained similarly. **Q.E.D.**

To demonstrate how Lemma 2.3 can be used, consider the three cases shown in Figure 2.6, where m = 3. The D-vectors of attaching and attached nodes can be easily obtained as follows.

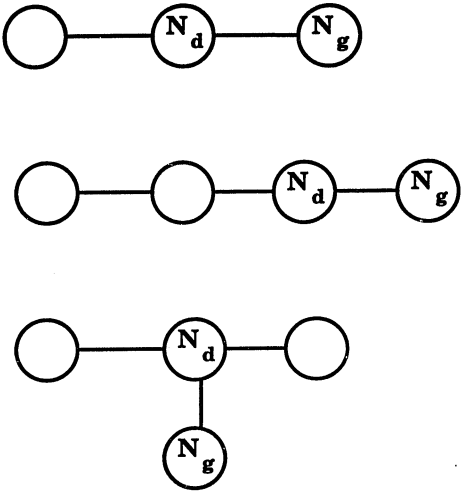


Figure 2.6. Three cases of adding a node N_g to a node N_d.

(i) $D_d' = \begin{bmatrix} 2 \\ 3 \\ 3 \\ 2 \end{bmatrix}$ from Figure 2.5. $\Rightarrow D_d = \begin{bmatrix} 4 \\ 9 \\ 9 \\ 4 \end{bmatrix}$ and $D_g = \begin{bmatrix} 5 \\ 8 \\ 8 \\ 5 \end{bmatrix}$

(ii) $D_d' = \begin{bmatrix} 5 \\ 8 \\ 8 \\ 5 \end{bmatrix}$ from D_g of (i) $\Rightarrow D_d = \begin{bmatrix} 10 \\ 24 \\ 24 \\ 10 \end{bmatrix}$ and $D_g = \begin{bmatrix} 13 \\ 21 \\ 21 \\ 13 \end{bmatrix}$

(iii) $D_d' = \begin{bmatrix} 4 \\ 9 \\ 9 \\ 4 \end{bmatrix}$ from D_d of (i) $\Rightarrow D_d = \begin{bmatrix} 8 \\ 27 \\ 27 \\ 8 \end{bmatrix}$ and $D_g = \begin{bmatrix} 12 \\ 22 \\ 22 \\ 12 \end{bmatrix}$.

Note that for any node in the graph the sum of entries in its D-vector represents the number of assignments. It is also easy to see that $\sum_{i=0}^{m} D_d(i) = \sum_{i=0}^{m} D_g(i)$. Using D-vectors, we can determine the bounds of the number of assignments by the theorem below.

Theorem 2.5: The number of assignments from the integer set I_m to any connected graph with p nodes, subject to the constraint that the difference between the numbers assigned to two adjacent nodes must be less than or equal to one, lies within the interval $[m(2^p-1)+1, 2^p+3^{p-1}(m-1)]$, where $m \geq 1$.

Proof: Obviously, the number of acceptable assignments for any connected graph is always less than or equal to that of its spanning tree. That is, the upper bound is attained by a tree structure. Now, we want to prove that the maximum is attained when the tree is a star structure, and then the upper bound follows from Lemma 2.3.

Since $\sum_{k=0}^{m} D_i(k)$ is the same for every N_i in a tree T, let $N(T,m) \equiv \sum_{k=0}^{m} D_i(k)$. For convenience, a tree T with p nodes is said to satisfy the *C-property*, if $N(T, m) \leq 2^p+3^{p-1}(m-1)$. Clearly, the C-property is satisfied by every tree with p nodes when $p \leq 3$. Consider the case

when one more node N_g is attached to a tree T with the C-property. Let D_d and D_d' denote respectively the D-vectors of N_d in the resulting and the original trees. Note that from Lemma 2.3 we have $D_d(i) \le 3^{p-1}$ $\forall$ $N_i \in V(N)$, and thus $\sum_{i=1}^{m-1} D_d(i) \le 3^{p-1}(m-1)$. Since T satisfies the C-property, we get $\sum_{i=1}^{m-1} D_d(i) + D_d(0) + D_d(m) \le 3^{p-1}(m-1) + 2^p$, which, by (b) of Lemma 2.3, leads to $\sum_{i=0}^{m} D_d'(i) = (\sum_{i=1}^{m-1} D_d(i))*3 + (D_d(0) + D_d(m))*2 \le 3^p(m-1) + 2^{p+1}$. This means that the resulting tree also satisfies the C-property, and the upper bound thus follows by induction.

Consider the lower bound. Since the complete graph K_p with p nodes possesses the maximal number of edges among all the graphs with p nodes, K_p attains the minimal number of assignments. Note that there are at most two distinct numbers which may occur in each assignment to K_p, and their difference must be less than or equal to one. There are 2^p ways to assign the numbers in the pair {j, j−1} to p nodes, where $1 \le j \le m$. Assignment of the same number to every node, say j, occurs both in the case of {j+1, j} and {j, j−1}, where $1 \le j \le m-1$. Thus, the total number of assignments is obtained by adding up the number of assignments from each pair {j, j−1}, where $1 \le j \le m$, to p nodes and subtracting double counts. Then, we get $2^p m - (m-1) = (2^p-1)m + 1$ for the lower bound. **Q.E.D.**

By Theorem 2.5, for a given network with p nodes the number of configurations to be evaluated must be within the interval $[n(2^p-1)+1, 2^p+3^{p-1}(m-1)]$, where $n = \min_{1 \le i \le p} \left\{O_i^*\right\}$ and $m = \max_{1 \le i \le p} \left\{O_i^*\right\}$. Note that due to the special nature of our problem, for a given topology a network with a higher average order of loop-free strategy does not always possess more configurations to be evaluated than the one with a lower average order of loop-free strategy.

An example is shown in Figure 2.7, where the network A has a higher average order of loop-free strategy than the network B, but B has more configurations to be evaluated. This is the very reason why $\max_{1 \le i \le p} \left\{ O_i^* \right\}$ and $\min_{1 \le i \le p} \left\{ O_i^* \right\}$ have to be used for upper and lower bounds, respectively.

Example 2.3: Consider the example network in Figure 2.8. Following the same procedure shown in the part (a) of Example 2.2, we can obtain [1, 1, 1, 1] as the minimal order vector of loop-free routing strategies. Clearly, there are $2^4 = 16$ possible configurations in this network.

As discussed in Section 2.2.2.1, the operational overhead required per second for the n-th order routing strategy, $R_c(n)$, can be assumed to have the form of $an + b$. For the sake of a numerical demonstration, let $a = 2.1$, $b = 1.2$ and C_α, C_β, C_γ be the configurations with strategy vectors [1, 0, 0, 0], [1, 1, 0, 0] and [1, 1, 0, 1], respectively.

(a). $RC(C_\alpha)$, $RC(C_\beta)$ and $RC(C_\gamma)$ are obtained from Eq. (2.14) as follows.

$RC(C_\alpha) = R_c(1) + 3R_c(0) = 6.9$

$RC(C_\beta) = 2R_c(1) + 2R_c(0) = 9$

$RC(C_\gamma) = 3R_c(1) + R_c(0) = 11.1$

(b). $RT(C_\alpha)$, $RT(C_\beta)$ and $RT(C_\gamma)$ can be determined as follows.

(i). With configuration C_α whose strategy vector is [1, 0, 0, 0], we find

$$SPL(C_\alpha) = \left\{ (N_2, N_3, N_2), (N_3, N_4, N_3), (N_3, N_2, N_3), (N_1, N_2, N_1) \right\}.$$

Using Eq. (2.19), we can obtain $m_{u,v-i,j}(C_\alpha)$ as follows: $m_{1,4-1,4}(C_\alpha)=2$, $m_{2,1-2,1}(C_\alpha)=2$, $m_{3,1-2,1}(C_\alpha)=1$, $m_{1,3-2,3}(C_\alpha)=1$, $m_{3,2-3,2}(C_\alpha)=2$, $m_{4,2-3,2}(C_\alpha)=1$, $m_{2,4-3,4}(C_\alpha)=1$, $m_{3,4-3,4}(C_\alpha)=2$,

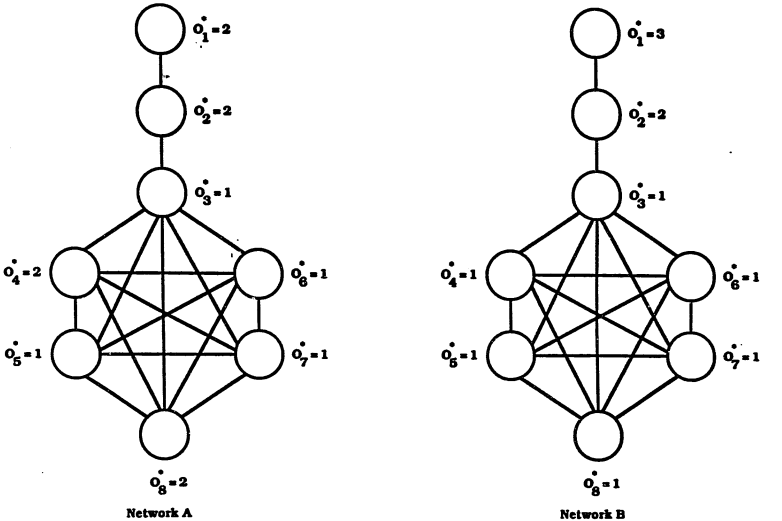


Figure 2.7. Two comparative example networks where A has a higher average order of looping-free strategy, whereas B has more configurations to be evaluated.

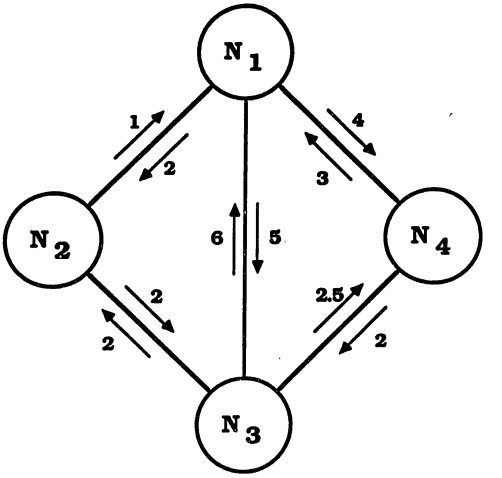


Figure 2.8. A network for Example 2.3.

and $m_{u,v-i,j}(C_\alpha)=0$ elsewhere.

Then, from Eq. (7) we get

$$\begin{cases} Rt(L_{14};\, C_\alpha) = \frac{1}{6} \\ Rt(L_{21};\, C_\alpha) = \frac{1}{4} \\ Rt(L_{23};\, C_\alpha) = \frac{1}{12} \end{cases} \Rightarrow RT(C_\alpha) = \left(\frac{\frac{1}{6}+\frac{1}{4}+\frac{1}{12}+\frac{1}{4}+\frac{1}{4}}{10}\right) = \frac{1}{10}.$$

$$\begin{cases} Rt(L_{32};\, C_\alpha) = \frac{1}{4} \\ Rt(L_{34};\, C_\alpha) = \frac{1}{4} \end{cases}$$

(ii). With the configuration C_β whose strategy vector is [1, 1, 0, 0], we get

$$SPL(C_\beta) = \left\{(N_2, N_3, N_2), (N_3, N_4, N_3)\right\}.$$

From Eq. (2.19), we can obtain $m_{u,v-i,j}(C_\beta)$ as follows: $m_{2,1-2,1}(C_\beta)=2$, $m_{3,1-2,1}(C_\beta)=1$, $m_{1,3-2,3}(C_\beta)=1$, $m_{3,2-3,2}(C_\beta)=2$, $m_{4,2-3,2}(C_\beta)=1$, $m_{2,4-3,4}(C_\beta)=1$, and $m_{u,v-i,j}(C_\beta)=0$ elsewhere.

Then, by Eq.(7), we get $Rt(L_{21};\, C_\beta) = \frac{1}{4}$, $Rt(L_{23};\, C_\beta) = \frac{1}{12}$, $Rt(L_{32};\, C_\beta) = \frac{1}{4}$, $Rt(L_{34};\, C_\beta) = \frac{1}{12}$ and $RT(C_\beta) = \frac{1}{15}$.

(iii). With configuration C_γ, whose strategy vector is [1, 1, 0, 1], following the same procedure, we obtain $SPL(C_\gamma) = \left\{(N_2, N_3, N_2)\right\}$, and then $RT(C_\gamma) = \frac{1}{20}$.

(c). Suppose our design objective is to minimize the function $g(C_k) = RC(C_k) + \lambda RT(C_k)$, where λ is the weighting factor between the network adaptability and operational overhead in handling routing messages. Note that $F(C_k) = \frac{1}{g(C_k)}$ in this case. If we choose λ to be 120,

then we get[2]

$$g(C_\alpha) = 6.9 + \frac{120}{10} = 18.9,$$

$$g(C_\beta) = 9 + \frac{120}{15} = 17,$$

$$g(C_\gamma) = 11.1 + \frac{120}{20} = 17.1.$$

When the above objective function is used, the configuration with a strategy vector [1, 1, 0, 0] is better than those with strategy vectors [1, 0, 0, 0] and [1, 1, 0, 1]. Therefore, from this procedure we can determine the optimal configuration of this network.

2.2.3. Remarks

Using the procedure discussed thus far, one can determine the optimal configuration from a given network topology and its link delays. The minimal order routing strategy for each node can be used to indicate how to construct a routing routing message for the node in order to avoid looping. Although the analytical results we developed are for the case of single link failure, it can be verified that our approach of using the high order routing strategies is applicable to the case of multiple link/node failures to speed up the network recovery process. It is worth mentioning that the order of loop-free routing strategy for each node is determined from the number of links on a certain loop around that node, and may vary if link delays in the vicinity of the node change drastically within a short time period. A sudden, drastic change in link delays may force some nodes to alter their optimal paths. In such a case, new minimal order routing strategies for these nodes must be derived. This usually introduces significant overheads, thus making it practically unacceptable.

[2]This choice is arbitrary and does not affect our method but yields interesting solutions.

However, in light of the derivation of Theorem 2.4, it can be verified that a higher-order loop is less likely to occur, since the delay of the higher-order loop is unlikely to be less than that of a second optimal path. Moreover, as we formulated in [78] and illustrated in Tables 2.1, 2.2 and 2.3, recovery from a link/node failure is sped up significantly when the order of routing strategy is increased; this is true even if the order of routing strategy is increased not so high as that derived from Corollary 2.4.1. Considering the above observations, one can determine the minimal order of routing strategy off-line, incorporate it into each node's routing strategy, and ignore small on-line changes in link delays. This will remove the necessity of on-line recalculation of minimal order routing strategies while allowing for acceptably fast recovery from node/link failures.

CHAPTER 3

ROUTING IN HEXAGONAL MESH MULTICOMPUTERS

Multicomputer interconnection networks are often required to connect thousands of homogeneously replicated processor-memory pairs [41, 44], each of which is called a *processing node* (PN). Instead of using a shared memory, all synchronization and communication between PNs for program execution is often done via message passing. Design and use of multicomputer interconnection networks have recently drawn considerable attention due to the availability of inexpensive, powerful microprocessors and memory chips [1, 26, 84, 92, 94]. The homogeneity of PNs and the interconnection network is very important because it allows for cost/performance benefits from the inexpensive replication of multicomputer components [63, 82]. Each PN in the multicomputer is preferred to have fixed connectivity so that standard VLSI chips can be used. Also, the interconnection network needs to contain a reasonably high degree of redundancy so that alternative routes can be made available to detour faulty nodes/links. More importantly, the interconnection network must facilitate efficient routing and broadcasting so as to achieve high performance in job executions.

In this chapter, hexagonal meshes (or H-meshes) are presented as a candidate multicomputer architecture that possesses all the foregoing salient features. As mentioned before, a large number of data manipulation applications requires the PNs on the hexagonal periphery to be wrapped around to achieve regularity and homogeneity such that identical software and protocols can be applied uniformly over the network. Clearly, the way of wrapping determines the topological structure of an H-mesh. For example, Martin proposed a general

method for forming a "boundary-less" topology for a large, dense, and regular arrangement of processor and memory modules on a two-dimensional surface, called a *processing surface* [57]. He showed specifically how to form such a topology on a square mesh and implied, without any elaboration, that a similar approach could be applied to different processing surfaces. Stevens illustrated an interesting way of wrapping the H-meshes of size less than or equal to three, where the size of an H-mesh is defined as the number of nodes on each peripheral edge of the H-mesh [82]. However, while the homogeneity of the H-mesh is not explored, his method for addressing, message routing and broadcasting in H-meshes is very complex and inefficient.

Consequently, in this chapter we propose a systematic method for wrapping H-meshes of arbitrary size as well as a new addressing scheme for the H-meshes. Efficient routing and broadcasting algorithms under the new addressing scheme will also be developed. As we shall see, the proposed addressing scheme not only achieves the homogeneity of PNs but also facilitates routing and broadcasting in the H-meshes significantly.

This chapter is organized as follows. In Section 3.1.1, a systematic way to wrap the H-mesh, called the *continuous type* (*C-type*) wrapping, is presented. Topological properties of H-meshes are explored in Section 3.1.2. In light of these properties, an addressing scheme for a C-type wrapped H-mesh is proposed in Section 3.2. With that addressing scheme, efficient routing and broadcasting algorithms are then developed. In Section 3.3, we compare the C-type wrapped H-meshes with some other existing multicomputer structures, such as hypercubes, trees, and square meshes. The results developed in the chapter have also been reported in [14].

3.1. Topology of C-Wrapped Hexagonal Meshes

In this section, a systematic method to wrap the H-mesh, is presented in Section 3.1.1 as a means of achieving the regularity and the homogeneity of PNs and several topological properties of H-meshes are derived in Section 3.1.2.

3.1.1. Systematic wrapping of hexagonal meshes

A graph is said to be *regular* if all the nodes in the graph have the same degree, and *homogeneous* if all the nodes in that graph are topologically identical. Clearly, homogeneity implies regularity, but the converse does not always hold. For example, the graph in Figure 3.1 is regular, but not homogeneous since node x and node y are not topologically identical. The degree of all nodes in an H-mesh without wrapping, except those on its periphery, is 6. Thus, such an H-mesh is neither homogeneous nor regular. Figure 3.2 illustrates an unwrapped H-mesh of size 3.

Consider the central node of the H-mesh in Figure 3.3. The node has six oriented directions, each of which leads to one of its six nearest neighbors. Without loss of generality, any of the six directions can be defined as the *x direction*, the direction 60 degrees clockwise to the x direction as the *y direction*, and then the direction 60 degrees clockwise to the y direction as the *z direction*. Once the x, y and z directions are defined, an H-mesh of size n can be partitioned into 2n–1 rows with respect to *any* of these three directions; there are three different ways of partitioning the rows of an H-mesh. Figure 3.3 shows the rows in an H-mesh of size 3 which are partitioned with respect to the x, y, and z directions, respectively. To facilitate our presentation, when an H-mesh of size n is partitioned into 2n–1 rows with respect to any of the three directions and the H-mesh is rotated in such a way that the corresponding direction from the central node points to the right, the top row is referred to as

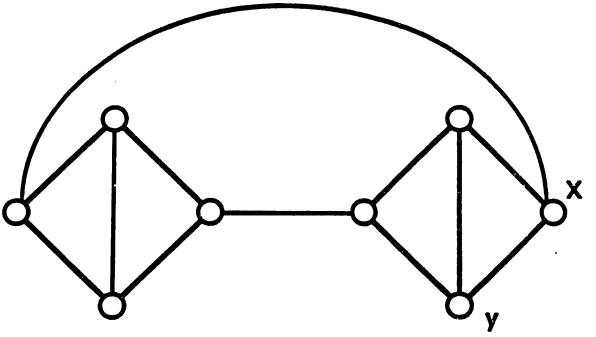


Figure 3.1. A regular non-homogeneous graph.

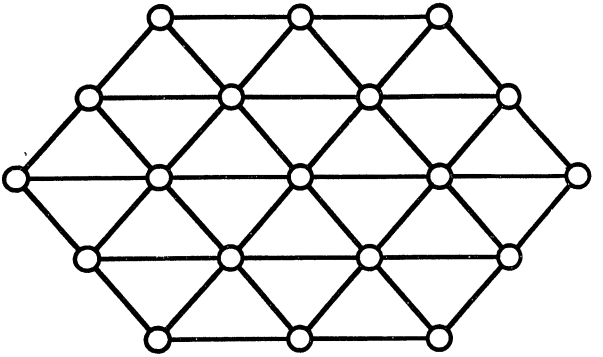


Figure 3.2. An H-mesh of size 3 without wrapping.

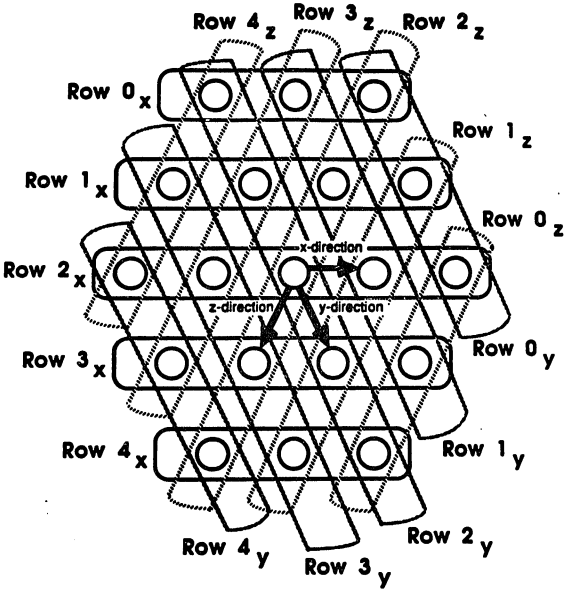


Figure 3.3. Partitioned rows of an H_3 in x, y and z directions.

row 0 in that direction, the second to the top row is referred to as row 1, and so on. Also, row n–1 is called the *central* row and rows 0 to n–2 and rows n to 2n–2 will sometimes be referred to as the *upper* and *lower* parts of an H-mesh of size n, respectively. (Subscripts for the row numbers in Figure 3.3 are used to indicate their directions.)

For notational simplicity, let $[b]_n \equiv b \bmod a$, for all $a \in \mathbf{I}^+$ and $b \in \mathbf{I}$, where $\mathbf{I}$ is the set of integers and $\mathbf{I}^+$ the set of positive integers. To make the H-mesh homogeneous and regular, the following method for wrapping an H-mesh continuously, called the *C-type wrapping*, is proposed.

C-type wrapping: Wrap an H-mesh of size n in such a way that for each of the three ways of partitioning the PNs into rows, the last PN in the row i is connected to the first PN of row $[i+n-1]_{2n-1}$.

As will be stated later in Corollary 3.1.1, the PNs in an H-mesh with the C-type wrapping are homogeneous. Furthermore, the C-type wrapping allows for an easy addressing scheme and, thus, simplifies the routing and broadcasting in an H-mesh significantly. In what follows, an H-mesh of size n with the C-type wrapping is denoted by H_n, while that without wrapping is denoted by H_n'.

The edges in the rows partitioned with respect to the x (y or z) direction and the associated wrapping are called the *edges in the x (y or z) direction*. An illustrative example of partitioning edges in an H_3 into three different directions is given in Figure 3.4 where the edges in each direction are drawn separately for clarity. From Figures 3.3 and 3.4, it can be observed that there is a unique path from one node to another in an H_n along each of the three directions.

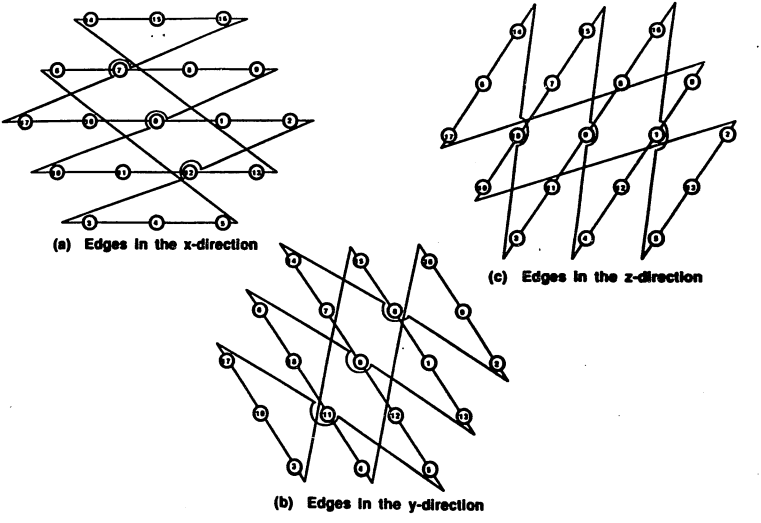


Figure 3.4. An H_3 with the C-type wrapping.

3.1.2. Topological properties of hexagonal meshes

The following two lemmas are direct results of the structure of an H-mesh.

Lemma 3.1: The number of nodes in an H_n is $p = 3n^2-3n+1$.

Proof: Since there are always n+i nodes in row i as well as in row 2n–i–2 for $0 \le i \le n-2$, and 2n–1 nodes in row n–1, the desired result follows from $\sum_{i=0}^{n-2} 2(n+i) + 2n-1 = 2n(n-1) + (n-1)(n-2) + 2n-1 = 3n^2-3n+1$. **Q.E.D.**

Lemma 3.2: The sum of the number of nodes in row i and that of row i+n is 3n–2 for $0 \le i \le n-2$.

The proof of Lemma 3.2 is trivial and, thus, omitted.

To exploit the topological properties of an H_n, each node is labeled with a 3-tuple as follows. Start from the central node of the H_n and label that node with (0, 0, 0), where the first coordinate of a node is referred to as its x labeling and the second and third coordinates are referred to as its y and z labelings, respectively. Then, move to the next node along the x direction, assign that node the x labeling one more than that of its preceding node, and so on. The y and z labelings for each node are determined by moving along the y and z directions respectively instead of the x direction. An example for the 3-tuple labeling of an H_3 is given in Figure 3.5, where the edges are not drawn for clarity.

Theorem 3.1: Let (x_1, y_1, z_1) and (x_2, y_2, z_2) be respectively the labelings of nodes n_1 and n_2 in an H_n and $p = 3n^2 - 3n + 1$. Then,

(i). $[x_2-x_1]_p = [(3n^2-6n+3)(y_2-y_1)]_p$.

(ii). $[x_2-x_1]_p = [(3n^2-6n+2)(z_2-z_1)]_p$.

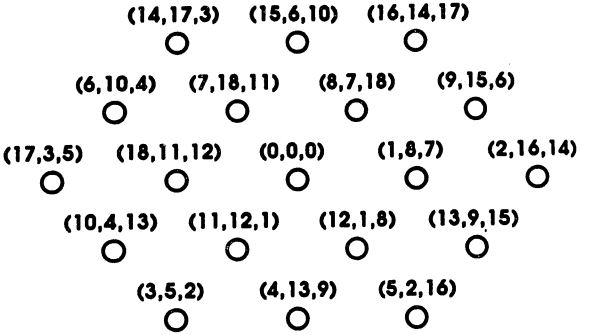


Figure 3.5. An H_3 with 3-tuple labeling.

Proof: For any pair of adjacent nodes n_u and n_v in the y-direction respectively with the labelings (x_u, y, z_u) and $(x_v, [y+1]_p, z_v)$, we want to claim $[x_v - x_u]_p = 3n^2-6n+3$. We shall prove this claim first and then (i) follows immediately from the claim.

Consider a path P^* from n_u to n_v following the +x direction *only*. (As can be seen in Figure 3.4a, such a path is unique.) Suppose n_u is in row r_u according to the row partition with respect to the x direction. Recall that the C-type wrapping requires the end of a row to be connected to the beginning of a row that is n-1 rows away from it. Since $[(n-1)(2n-3)]_{2n-1}=1$, P^* must run through nodes in 2n–2 different rows to get from n_u to n_v -- which are adjacent in the y direction -- including the rows that contain n_u and n_v. In other words, the path P^* must visit all but:

(1). those nodes ahead of n_u in row r_u,

(2). those nodes behind n_v in row $[r_u+1]_{2n-1}$, and

(3). those nodes in row $[r_u+n]_{2n-1}$ (the only row that P^* does not travel).

Note that if n_u is in the upper part or central row of an H_n, the total number of nodes in (1) and (2) will be one less than the number of nodes in row r_u. On the other hand, if n_u is in the lower part of the H_n, the total number of nodes in (1) and (2) will be one less than the number of nodes in row $[r_u+1]_{2n-1}$. By Lemma 3.2, we conclude that for both cases, the total number of nodes in (1), (2) and (3) is 3n–3, i.e., one less than 3n–2. Thus, the number of nodes on P^* is $(3n^2-3n+1) - (3n-3) = 3n^2-6n+4$. Since both n_u and n_v are contained in P^*, the claim is thus proved and (i) follows.

For (ii), we claim that for any pair of adjacent nodes n_u and n_v in the z-direction labeled respectively with (x_u, y_u, z) and $(x_v, y_v, [z+1]_p)$, $[x_v - x_u]_p = 3n^2-6n+2$. This claim can be proved by following the same logic as the above and, thus, (ii) can be proved similarly.

Q.E.D.

Note that $(3n^2-3n+1) - (3n^2-6n+2) = 3n-1$, $(3n^2-3n+1) - (3n^2-6n+3) = 3n-2$. In light of Theorem 3.1, an H_n can be redrawn as a power cycle with $p=3n^2-3n+1$ nodes, in which node i is not only adjacent to nodes $[i-1]_p$ and $[i+1]_p$, but also adjacent to nodes $[i+3n-1]_p$, $[i+3n-2]_p$, $[i+3n^2-6n+2]_p$ and $[i+3n^2-6n+3]_p$. For example, an H_3 can be redrawn as the one in Figure 3.6. This fact leads to the following corollary.

Corollary 3.1.1: All the processing nodes in an H_n are homogeneous.

Note that, although extensive results have been obtained for many classes of networks [90, 87, 36, 7], the connection pattern in an H-mesh does not belong to any of previously found patterns.

Lemma 3.3:

(i). The number of links in an H_n' is $9n^2-15n+6$.

(ii). The number of links in an H_n is $9n^2-9n+3$.

Proof: We prove (ii) first. From the fact that the summation of all node degrees in a graph is twice the number of edges in that graph, we have $6(3n^2-3n+1)/2 = 9n^2-9n+3$.

Since there are $6(n-1)$ nodes in the periphery of an H_n', six of which have degree 3 and $6n-12$ of which have degree 4, we have the summation of all node degrees in an H_n': $6(3n^2-3n+1-6n+6) + 6*3 + (6n-12)4 = 18n^2-30n+12$, making the number of links in an H_n' equal to $9n^2-15n+6$. **Q.E.D.**

Lemma 3.4:

(i). The diameter in an H_n is $n-1$.

(ii). The average distance in an H_n is $(2n-1)/3$.

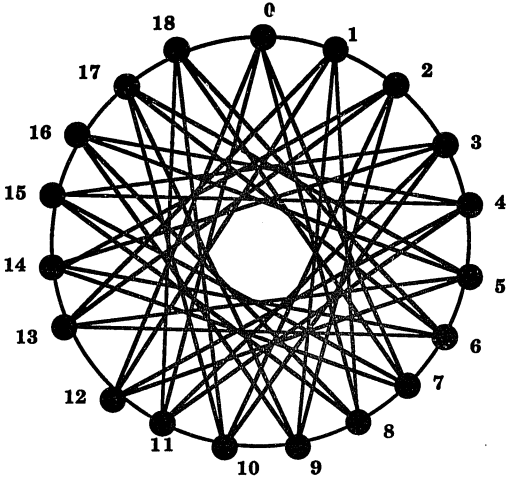


Figure 3.6. A redrawn H_3.

Proof: (i) follows from the fact that, without loss of generality, every node in a wrapped H-mesh can view itself as the central node of the mesh. To prove (ii), let $td(H_n)$ denote the total summation of distances from any node to all the other nodes in an H_n. Then,

$$td(H_n) = \sum_{d=1}^{n-1} 6d^2 = 6\frac{n(n-1)(2n-1)}{6} = n(n-1)(2n-1).$$

The average distance is thus $td(H_n)/(p-1) = (2n-1)/3$. **Q.E.D.**

Moreover, Lemma 3.1 and (i) of Lemma 3.4 lead to the following lemma.

Lemma 3.5: The diameter of an H-mesh with p nodes grows as $O(p^{1/2})$.

In addition, it is worth mentioning that H-meshes possess a high degree of fault-tolerance measured in terms of *connectivity*. Recall that a graph is said to be *m-connected* (*m-edge connected*) if m is the minimal number of nodes (edges) whose removal will disconnect the graph [92]. It can be easily verified that an H-mesh of any size is 6-connected and also 6-edge connected. This means that an H-mesh can tolerate up to five node/link failures at a time, which is better than most of the other existing networks [93, 37].

3.2. Routing and Broadcasting in Hexagonal Meshes

In this section, a simple addressing scheme for H-meshes is developed on the basis of the C-type wrapping. Under this addressing scheme, shortest paths from one node to any other node can be computed by the source node with an extremely simple algorithm of $O(1)$. A point-to-point broadcasting algorithm is also developed, which is proved to be optimal in the number of required communication steps.

3.2.1. Addressing scheme

Using Theorem 3.1, the y and z labelings of any node can be obtained from its x labeling, meaning that only one (instead of three) labeling will suffice to uniquely identify any node in an H-mesh. An example of this addressing for an H_4 is given in Figure 3.7a where all edges are omitted for clarity. This addressing scheme is much simpler than the one in [82], since only one number, instead of two, is needed to identify each node in an H_n. Furthermore, the routing strategy under this addressing scheme will be shown to be far more efficient than the one in [82], especially when messages must be routed via wrapped links.

3.2.2. Routing

Under the above addressing scheme, a shortest path between any two nodes can be easily determined by the difference of their addresses.

Let m_x, m_y, and m_z be respectively the numbers of moves or hops from the source node to the destination node along the x, y and z directions on a shortest path. Negative values mean the moves are in opposite directions. Note that there could be several equally short paths from a node to another, and these shortest paths are completely specified by the values of m_x, m_y and m_z. More precisely, it can be verified that for i = x, y, z, the number of paths with m_i moves in the corresponding directions is $\frac{(|m_x| + |m_y| + |m_z|)!}{|m_x|!\,|m_y|!\,|m_z|!}$. Let s and d be respectively the addresses of the source and destination nodes, and $k = [d-s]_p$ where $p = 3n^2-3n+1$. Then, the m_x, m_y, and m_z for shortest paths from s to d can be determined by the algorithm given below.

Algorithm A_1

begin

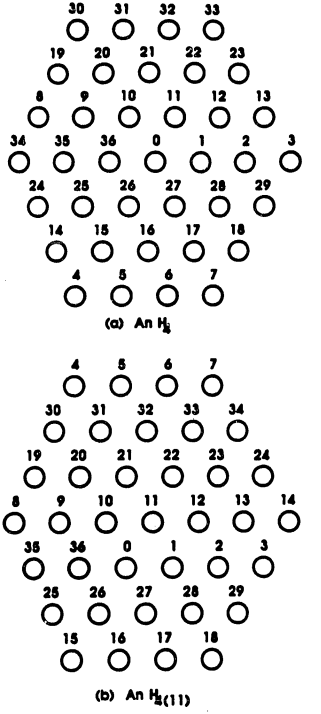


Figure 3.7. An illustrative example of Theorem 3.2.

```
    m_x := 0;  m_y := 0;  m_z := 0;
    if (k < n) then begin m_x := k;  stop end;
    if (k > 3n^2–4n+1) then begin m_x := 3n^2–3n+1–k;  stop end;
    r := (k–n) div (3n–2);
    t := (k–n) mod (3n–2);
    if (t ≤ n+r–1)
      then /* d is in the lower part of the H-mesh centered at s. */
          if (t ≤ r)
            then m_x := t–r;  m_z := n–r–1;
            else if (t ≥ n–1) then m_x := t–n+1; m_y := n–r–1;
                              else m_y := t–r;  m_z := n–t–1;
                endif;
          endif;
      else /* d is in the upper part of the H-mesh centered at s. */
          if (t ≤ 2n–2)
            then m_x := t+2–2n; m_y := –r–1;
            else if (t ≥ 2n+r–1) then m_x := t–2n–r+1;  m_z := –r–1;
                                 else m_y := t+1–2n–r;  m_z := 2n–t–2;
                endif;
          endif;
    endif;
    stop
end;
```

The correctness of A_1 is proved by the theorem below.

Theorem 3.2: The values of m_x, m_y and m_z determined by A_1 completely specify all the shortest paths from s to d in an H_n.

Proof: By Theorem 3.1, without loss of generality the source node can view itself as the central node of the H_n. Let $H_{n(s)}$ be the H-mesh centered at s. For the case when d is in the central row (i.e., row n–1) of $H_{n(s)}$, m_x is determined by the statements in lines 2 and 3 of A_1, and m_y=m_z=0.

Consider the case when d is in a row other than row n–1 of $H_{n(s)}$. Form a *group* of nodes with the nodes of rows n–i–2 and 2n–i–2: Call this *group i*. Then, in Figure 3.3 rows 1

and 4 form group 0, and rows 0 and 3 form group 1. A group consists of two rows, one from the upper part and the other from the lower part of $H_{n(s)}$. We know from Lemma 3.2 that each group contains 3n–2 nodes. Then, by the statements in lines 4 and 5 of A_1, r is determined as the identity of the group which contains d in $H_{n(s)}$, and t is the position of d in group r. We can determine from t which row of $H_{n(s)}$ contains d. Denote the first node of that row by n_f. A shortest path from s to n_f and that from n_f to d can thus be determined. Using the idea of the composition of vectors, we get the desired equations for m_x, m_y, and m_z.

Q.E.D.

An illustrative example for routing in an H_4 is given in Figure 3.7 where edges are omitted for clarity. Suppose node 11 sends a message to node 5, i.e., n=4, s=11, d=5 and $k = [5-11]_{37} = 31$. The original H_4 is given in Figure 3.7a and $H_{4(11)}$ is in Figure 3.7b, where node 11 is placed at the center of the H_4. (As mentioned before, this can be done without loss of generality.) From A_1, we get r=2, t=7 and then m_x=0, m_y=–2 and m_z=–1. Note that the route from node 11 to node 5 is isomorphic to that from node 0 to node 31. This is not a coincidence, but rather, a consequence of the homogeneity of H_4. All paths from one node to another in an H_n are completely determined by the difference in their addresses.

Moreover, the complexity of A_1 is $O(1)$, which is independent of the size of H-mesh, and the source node needs to execute the algorithm. Once m_x, m_y and m_z are determined, they form a routing record to be sent along with a regular message. The routing in an H_n is then characterized by the following six routing functions:

$$R_{m_x-1}(i) = [i + 1]_p$$

$$R_{m_x+1}(i) = [i - 1]_p$$

$$R_{m_y-1}(i) = [i + 3n^2 - 6n + 3]_p$$

$R_{m_y+1}(i) = [i + 3n - 2]_p$

$R_{m_z-1}(i) = [i + 3n^2 - 6n + 2]_p$

$R_{m_z+1}(i) = [i + 3n - 1]_p$,

where, as before, $p=3n^2 - 3n+1$ is the total number of nodes in an H_n.

The routing record is updated by the above six functions at each intermediate node on the way to the destination node so that the current routing record may contain the correct information for the remaining part of the path. The above functions are applied repeatedly until $m_x=m_y=m_z=0$, meaning that the message has reached the destination node.

3.2.3. Point-to-point broadcasting

Applications in various domains require an efficient method for broadcasting messages to every node in an H-mesh. Due to interconnection costs, it is very common to use point-to-point communications for broadcasting. In this subsection, we present a broadcasting algorithm using point-to-point communications[3]. Further, we shall prove that this algorithm is optimal in the number of required communication steps. We then apply this algorithm in developing a systematic procedure for computing the sum of numbers distributed across the nodes of an H-mesh.

Without loss of generality, we can assume the center node to be the source node of the broadcast. The set of nodes which have the same distance from the source node is called a *ring*. The main idea of our algorithm is to broadcast a message, ring by ring, toward the periphery of an H-mesh. The algorithm consists of two phases. In the first phase which takes 3 steps, the message is transmitted to the 6 nearest neighbors of the origin. As shown in Figure

[3]This broadcasting algorithm was proposed by D. D. Kandlur, in the University of Michigan, Ann Arbor.

3.8, node a sends the message along the +x direction to node b in step 1, nodes a and b send messages along the z direction to nodes c and d respectively in step 2, and then, nodes a, b and c send messages along the –y direction to nodes e, f and g respectively in step 3. At the end of this phase, nodes b, c, d, e, f and g are assigned the directions x, z, y, –y, –z, and –x respectively as the propagation direction.

Note that there are six corner nodes in each ring. In the second phase which takes n–1 steps, the six corner nodes of each ring send the message to two neighboring nodes respectively while all the other nodes propagate the message to the next node along the direction in which the message was previously sent. An illustrative example for the broadcasting in in Figure 3.9. A formal description of this procedure is given in algorithm A_2 below, where the five arguments in the *send* and *receive* commands denote respectively the message to be broadcast, the direction from which the message is received, the direction to propagate the message, a flag indicating whether it is a corner node or not, and the count of communication steps. Also, the function *rotate* is used for each corner node to determine the direction of the second transmission, in which the direction is rotated clockwise. For example, y= rotate(x, +60°), z= rotate(y, +60°) and –x= rotate(z, +60°).

Algorithm A_2

Procedure for the source node

```
begin
    send(message, x, x, TRUE, 1);
    send(message, z, z, TRUE, 2);
    send(message, –y, –y, TRUE, 3);
end
```

Procedure for all nodes except the source node

```
begin
    receive(message, from-dirn, propagate-dirn, corner, count);
    case count of
```

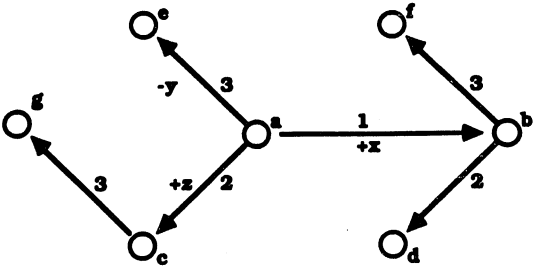


Figure 3.8. First phase of the broadcasting algorithm.

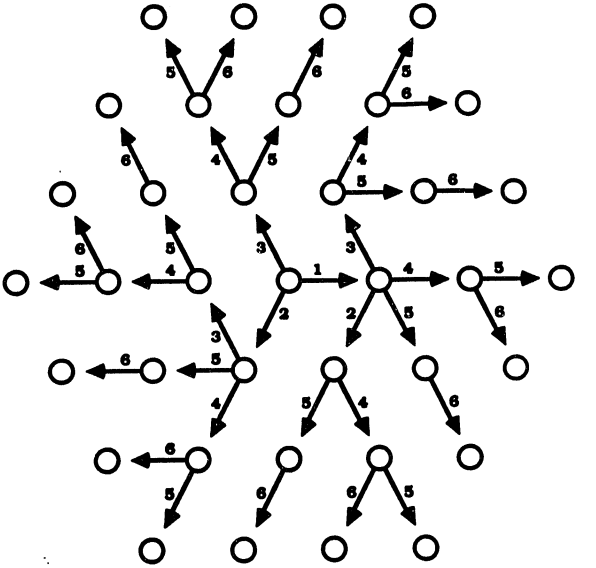


Figure 3.9. Broadcasting in an H_4 .

```
1:  begin
        send(message, z, y, TRUE, 2);
        send(message, –y, –z, TRUE, 3);
        count:=3;
    end
2:  begin
        if (from-dirn=z) then send(message, –y, –x, TRUE, 3);
        count:=3;
    end
n+2: stop        /* test for termination*/
else:                    /* do nothing*/
endcase
/* steps of the second phase */
if (corner) and (count ≤ n+1) then
begin
    direction-2 := rotate (propagate-dirn, +60°);
    send(message, propagate-dirn, propagate-dirn, TRUE, count+1);
    send(message, direction-2, direction-2, FALSE, count+2);
end
else send(message, propagate-dirn, propagate-dirn, FALSE, count+1);
end
```

Note that the second phase is required only when n>2. Thus, the total number of communication steps is n+2 when n≥3, and 3 when n=2. Furthermore, we prove by the following theorem that A_2 is an optimal broadcasting algorithm in the number of required communication steps.

Theorem 3.3: Any broadcast algorithm for the hexagonal mesh H_n, n≥3, which uses point-to-point communication requires at least n+2 communication steps.

Proof: We first prove the result for H-meshes of size 3 and 4. An H_3 has 19 nodes and so, even if recursive doubling were used for each step of broadcasting, at least 5 steps are required to cover all the nodes. A similar argument applies for an H_4 which has 37 nodes, i.e., at least 6 steps are needed for the H_4. For larger values of n, we show that it is not possible

to cover all nodes in an H_n in n+1 steps.

To show this, we examine all possible patterns of nodes which can be reached using two communication steps. In two steps, only four nodes can be reached and when the duplicate patterns arising due to various symmetries are removed, we get only the six unique patterns shown in Figure 3.10. These patterns can each be mapped onto the nodes on the periphery of an H_n, and in each case we can find at least five nodes which are n–1 links away from each of the four mapped nodes. Note that, at most four nodes of these five can be reached in n–1 steps from the original four nodes since only point-to-point communication is permitted. Hence, there is at least one node which cannot be reached in n+1 steps. Therefore, at least n+2 steps are required. Figure 3.11 shows the mappings of the six possible patterns into an H_5, but it is clear from the figure that a similar mapping would apply for all larger meshes, and the theorem thus follows. **Q.E.D.**

To see an application of the broadcasting algorithm, consider the problem of computing the sum of numbers distributed across the nodes in the H-meshes. Note that this problem is important, since it occurs frequently in many applications such as the computation of an inner product of two vectors. The global sum can be obtained by first computing the inner product within each node for the segments of the vectors in the node and then summing these partial sums up. Typically, for vector operations, the inner product is required by all nodes. In light of the broadcasting algorithm, the procedure of obtaining the global sum can be accomplished systematically by 2n+1 communication steps as described below.

The procedure can be divided into two phases. In the first phase, the partial sums are transmitted toward a center node, while each node along the path computes the sum of all incoming partial sums and its own number, and then transmits the resulting partial sum inwards. An illustrative example of the first phase in an H_4 is shown in Figure 3.12. It is

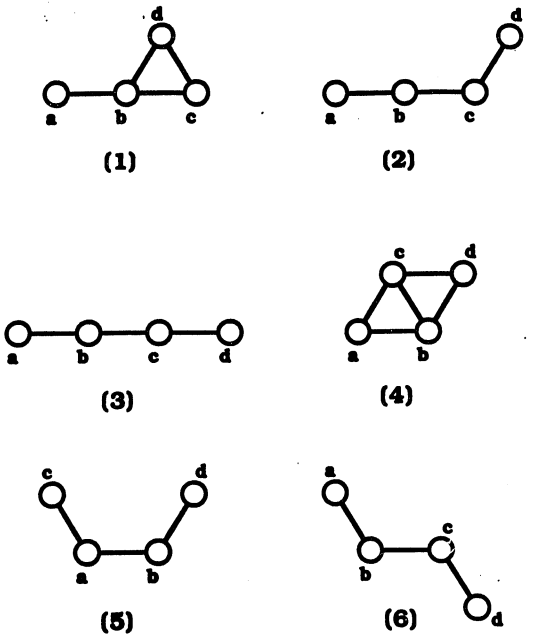


Figure 3.10. Six possible patterns after two communication steps.

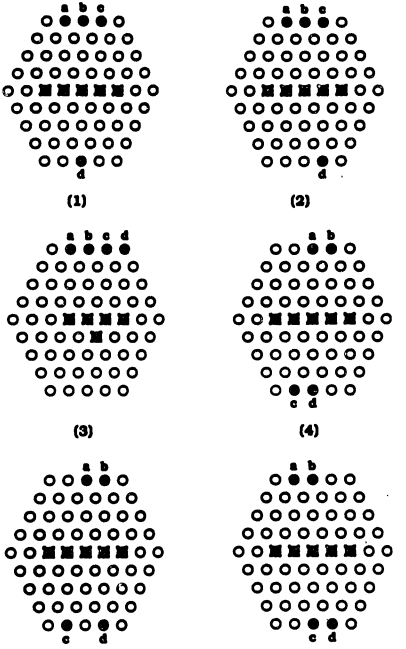


Figure 3.11. Embedding in an H_5 of the six patterns in Figure 3.10.

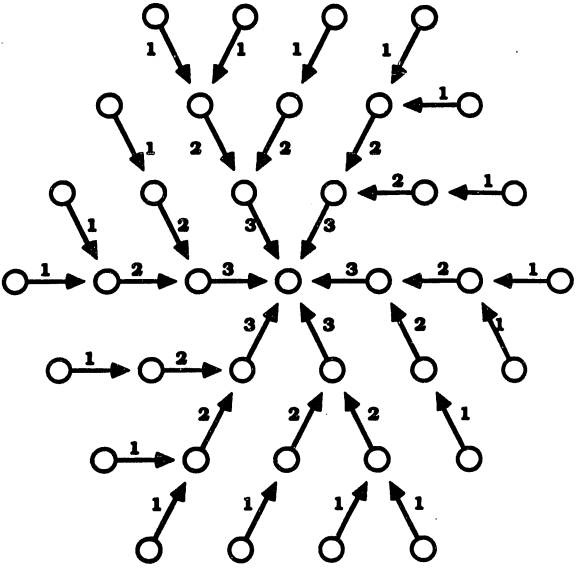


Figure 3.12. First phase of the global sum algorithm in an H_4.

easy to see that the first phase requires n−1 communication steps in an H_n. In the second phase, the center node, after adding the six incoming partial sums with its own number, uses the point-to-point broadcasting to distribute the sum to all nodes. Since we need n+2 steps in the second phase, the total number of required communication steps for obtaining the global sum is 2n+1.

3.3. Comparative Remarks on H-Meshes

It is essential that messages in a large multicomputer network be routed by each intermediate PN without using information about the entire network, since large storage and time overheads are required for maintaining such global information. As mentioned earlier, the C-type wrapping of an H-mesh not only provides regularity and homogeneity to the interconnection network but also allows for a very simple addressing scheme in the H-mesh, completely specifying all shortest paths from one node to any other node only with the addresses of the source-destination pair. Shortest paths can then be computed by the source node with an elegant algorithm of the complexity $O(1)$. This algorithm is better than the one commonly used for hypercube multicomputers of complexity $O(\log_2 p)$ that relies on an address comparison procedure [42], and those using routing tables of complexity $O(p)$, where p is the number of nodes in the system. In light of homogeneity, a systematic broadcasting algorithm, which is proved to be optimal in the number of communication steps, is also available.

Several important features of various architectures are compared with those of H-meshes and tabulated in Table 3.1, where connectivity is defined as the minimal number of nodes whose removal will disconnect the network and the O-notation is used to denote the complexity as a function of the network size. Although complete graphs and stars have constant diameters [38], they are not attractive because of the excessive number of connections

parameters / structures	node number	link number	diameter	node degree	connectivity	message density †
Complete graphs	p	p(p−1)/2	1	p − 1	p	$O(1/p)$
Hypercubes, Q_n	$p = 2^n$	$n2^{n-1}$	$O(\log_2 p)$	$\log_2 p$	$\log_2 p$	$O(1)$
k-level CBTs ††	$p = 2^k-1$	p − 1	$O(\log_2 p)$	root: 2 leaves: 1 others: 3	1	$O(\log_2 p)$
Stars	p	p − 1	2	center: p−1 branches: 1	1	center: p branches: 2
Cycles	p	p	$O(p)$	2	2	$O(p)$
Cordal rings	p	3p/2	$O(p^{1/2})$	3	3	$O(p^{1/2})$
Square meshes, w × w	$p = w^2$	2p	$O(p^{1/2})$	4	4	$O(p^{1/2})$
H-meshes, H_n	$p = 3n^2-3n+1$	3p	$O(p^{1/2})$	6	6	$O(p^{1/2})$

† Message density is measured in messages per link per time interval under the assumption that each node sends a message to every other node within one time interval.

†† CBT stands for a complete binary tree.

Table 3.1. Comparisons among various multicomputer architectures.

required for complete graphs and the poor reliability of stars. On the other hand, the diameter of a hypercube is allowed to grow as $O(\log_2 p)$ at the cost of increasing the node degree, which could cause a serious wiring problem with the expansion of the network (i.e., fan-in and fan-out problems) and may make it difficult to implement with standard VLSI chips. This is viewed as a major drawback of the hypercube structure [44].

Consider the cases when both the hypercube and hexagonal mesh architectures have approximately the same number of nodes. A 7-dimensional hypercube, denoted by Q_7, has 128 nodes, one more than that of an H_7. However, the node degree of a Q_7 is 7 and that of an H_7 is 6, while the diameter of an H_7 is 6, one less than that of a Q_7; that is, an H_7 can be favorably compared with a Q_7. It is interesting, however, to see that the diameter of a Q_{10} (a 10-dimensional hypercube) with 1024 nodes is 10, whereas that of an H_{19} with 1027 nodes is 18. This results from the fact that the node degree of hypercubes grows as $O(\log_2 p)$ while that of H-meshes remains constant (6), implying the existence of a tradeoff between the node degree and the communication diameter. Moreover, H-meshes have a finer scalability than hypercubes, i.e., $O(n^2)$ for H-meshes and $O(2^n)$ for hypercubes.

The complete binary tree structure offers the advantages of a constant node degree and an $O(\log_2 p)$ diameter. However, a tree is vulnerable to single link/node failures and suffers from the serious congestion of messages around its root [43]. These problems can be alleviated somewhat by adding additional links between leaf nodes. But the addition of links makes the optimal (shortest path) routing of messages very difficult and complicated [33]. Note that, although some other architectures such as chordal rings and square meshes [25, 2] also have a diameter of $O(p^{1/2})$ while their node degrees are less than that of an H_n, their lower connectivities will naturally lower the degree of fault-tolerance. This is undesirable, especially when applications require the multicomputer system to have a reasonably high degree of fault-

tolerance.

In general, an architecture strong in one aspect may be weak in the others; that is, no single architecture provides every desired feature. There are several types of architectures known to be suitable for interconnecting a large number of PNs, and H-meshes are one of them with many desirable features. Therefore, in view of their homogeneity, routing and broadcasting, fault-tolerance and implementability, the H-meshes with the C-type wrapping are an attractive candidate architecture for interconnecting a large number of PNs.

CHAPTER 4

ADAPTIVE FAULT-TOLERANT ROUTING IN HYPERCUBE MULTICOMPUTERS

In recent years, hypercube multicomputers have been drawing considerable attention due mainly to their structural regularity for easy construction and high potential for the parallel execution of various algorithms [21,91]. Numerous research efforts related to hypercube architectures, operating systems, programming languages, etc., have been undertaken [4,5,11,12,27], and several research (e.g., by Caltech [76]) and commercial (by Intel, NCUBE, Floating Point Systems, Ametek and Thinking Machine, to name a few) hypercube multicomputers have been built.

Efficient routing of messages is a key to the performance of any multiprocessor or multicomputer system. Especially, the increasing use of multiprocessor/multicomputer systems for reliability-critical applications has made it essential to design fault-tolerant routing strategies for such systems. By fault-tolerant routing, we mean the successful routing of messages between any pair of non-faulty nodes in the presence of faulty components (links and/or nodes). When hypercube multicomputers are to be used for reliable applications, they must be made capable of routing messages even in the presence of faulty components [17,34,53].

Recall that a connected hypercube with faulty components is called an *injured hypercube*, whereas a hypercube without faulty components is called a *regular hypercube*. Some results on improving the routing efficiency in regular hypercubes are reported in [42,85].

Routing in an incomplete hypercube is also shown to be straightforward [47]. In a regular hypercube, each intermediate node can determine the next hop of a message by examining the message's destination address and choosing, from all its neighboring nodes, the one which is closest to the destination. Clearly, this can be accomplished by aligning the address of the source node with that of the destination node from right to left bit-by-bit [74]. However, this scheme becomes invalid in an injured hypercube, since the message may be routed to a faulty component. In order to enable non-faulty nodes in an injured hypercube to communicate with one another, each node has to be equipped with enough information so as to route messages around the faulty components.

Using additional hardware called a *hyperswitch*, a best-first search algorithm for routing messages in a hypercube is developed in [17]. Several adaptive packet routing algorithms are also presented in [48, 89], which, however, are not meant to take full advantage of the special structure of a hypercube. In addition, various algorithms are proposed in [3, 52] to broadcast the information about faulty components to all the other nodes in a hypercube so that the faulty components can be bypassed during message routing. Clearly, if each node is equipped with the information on all faulty components, then it can always determine a shortest fault-free path to route the message as long as the source and destination nodes are connected. However, it is usually too costly (in space and time) to equip every node with the entire network information especially when the size of the hypercube is large. Hence, it is important to develop routing schemes which require each node to keep only the information essential for making correct routing decisions.

For the reasons above, we shall first develop a routing scheme based on depth-first search, in which each node is required to know only the condition (healthy or faulty) of its own links. The network information is added to the message as the message travels toward

the destination. This scheme is shown to be capable of routing messages between any pair of non-faulty nodes. More importantly, this scheme will be proved to be very powerful in that the probability of routing messages via shortest paths is very high.

Due to the absence of network information at each hypercube node, the paths chosen by the above scheme may not be the shortest. This, however, is very undesirable especially when a certain pair of nodes communicate with each other frequently. Clearly, the efficiency of routing (measured in terms of the length of a path chosen) can be improved if every non-faulty node is equipped with more network information than that on its own links only, such that the shortest path of each message to its destination can be foreseen, and the faulty components can be bypassed. In view of this fact, we shall present a routing algorithm based on the propagation of the component failure information. However, as will be seen, this algorithm is applicable only when the number of faulty components is less than n, thus limiting its applicability. In light of the idea presented in Chapter 2, a routing algorithm using the network delay table will be developed for an injured hypercube of an arbitrary number of faulty components. Observe that the abundant connections in a hypercube usually make routing decisions in a node unaffected by the failure of a component located far away from the node. This implies that the network delay table of each node does not have to keep information on those nodes located far away, and can still lead to the shortest path routing.

This chapter is organized as follows. An adaptive routing scheme using depth-first search will be presented in Section 4.1. Performance of this routing scheme will be analyzed and the extension of this approach will be explored. In Section 4.2, routing algorithms using global information to achieve the shortest path routing will be developed. We shall develop two routing schemes, based on the propagation of component failure information and the network delay tables respectively. Part of the results in Section 4.1 have also been reported in [15].

4.1. Routing with Information on Local Link Failures

The necessary notation and definitions are introduced first in Section 4.1.1. An adaptive routing scheme based on depth-first search is then presented in Section 4.1.2. Performance analysis and the extension of this routing scheme will be given in Sections 4.1.3 and 4.1.4, respectively.

4.1.1. Preliminaries

An n-dimensional hypercube (or n-cube or Q_n) is formally defined as follows.

Definition 4.1: An n-cube, Q_n, is defined recursively as follows.

(i) Q_0 is a trivial graph with one node, and

(ii) $Q_n = K_2 \times Q_{n-1}$,

where K_2 is the complete graph with two nodes, Q_0 is a trivial graph with one node and $\times$ is the product operation of two graphs [38].

Examples for Q_1, Q_2 and Q_3 can be found in Figure 4.1. It follows from Definition 4.1 that a Q_n contains 2^n nodes and $n2^{n-1}$ links since the degree of each node in a Q_n is n. Let $\sum$ be the ternary symbol set {0, 1, *}, where * is a *don't care* symbol. Every subcube in a Q_n can then be uniquely represented by a string of symbols in $\sum$. Such a string of ternary symbols is called the *address* of the corresponding subcube. For example, the address of the subcube Q_2 formed by nodes 0010, 0011, 0110 and 0111 in a Q_4 is 0*1*. Figure 4.2 shows a Q_2 with address 0*1* in a Q_4. Note that the number of *'s in the address of a subcube is the same as the dimension of that subcube. The rightmost coordinate of the address of a subcube will be referred to as *dimension 1*, and the second rightmost coordinate as *dimension 2*, and so on. For each hypercube node, the communication link in dimension i is called the *i-th link* of the node. For notational simplicity, each link is represented by a binary string with a '-'

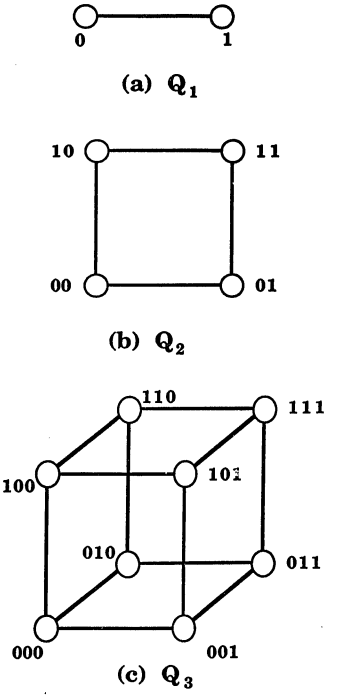


Figure 4.1. Illustrative examples of hypercubes.

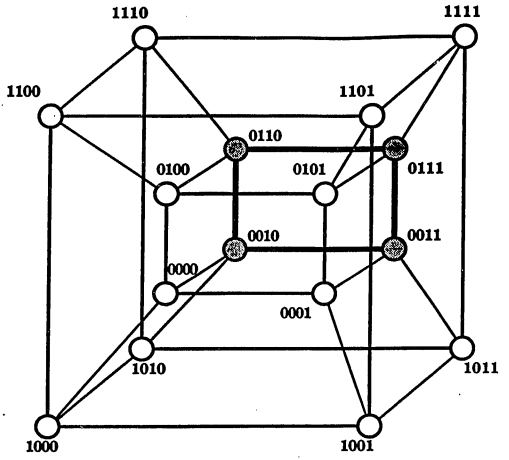


Figure 4.2. A Q_2 with address 0*1* in a Q_4.

symbol in the corresponding dimension. For example, the link between nodes 0000 and 0010 is represented by 00-0. Let r^+ and r^- denote the two end nodes of link r, where r^+ (r^-) represents the node whose address is obtained by changing the '-' symbol in the link's address to 0 (1).

Definition 4.2: The *Hamming distance* between two hypercube nodes with addresses $u = u_n u_{n-1} \cdots u_1$ and $w = w_n w_{n-1} \cdots w_1$ in a Q_n is defined as

$$H(u, w) = \sum_{i=1}^{n} h(u_i, w_i), \text{ where } h(u_i, w_i) = \begin{cases} 1 & \text{if } u_i \neq w_i, \\ 0 & \text{if } u_i = w_i. \end{cases}$$

For the nature of distributed routing strategies to be presented, it is necessary to introduce the definition of the *exclusive* operation between two binary strings, and the concept of *relative address* between two hypercube nodes.

Definition 4.3: The *exclusive* operation of two binary strings $q = q_n q_{n-1} \cdots q_1$ and $m = m_n m_{n-1} \cdots m_1$, denoted by $q \oplus m = r_n r_{n-1} \cdots r_1$, is defined as $r_i = 0$ if $q_i = m_i$ and $r_i = 1$ if $q_i = \overline{m_i}$ for $1 \leq i \leq n$.

Obviously, that the exclusive operation is commutative, i.e., $q \oplus m = m \oplus q$. We use $\oplus_{i=1}^{k}$ to denote a sequence of k exclusive operations. The relative address of a node u with respect to another node w, denoted by $u_{/w}$, can then be determined by $u_{/w} = u \oplus w$. The relative address of a subcube with respect to a node u can be determined by the relative addresses of all the nodes it contains. Let $e^k = e_n e_{n-1} \cdots e_1$ where $e_k = 1$ and $e_j = 0 \ \forall \ j \neq k$. For example, $1001 \oplus e^2 = 1011$, $0011_{/1001} = 1010$, and $0*1*_{/1001} = 1*1*$. Also, the *spanning subcube* of two hypercube nodes is defined as follows.

Definition 4.4: The spanning subcube of two nodes $u = u_n u_{n-1} \cdots u_1$ and $w = w_n w_{n-1} \cdots w_1$ in a Q_n, denoted by $SQ(u,w) = s_n s_{n-1} \cdots s_1$, is defined as $s_i = u_i$ if

$u_i = w_i$, and $s_i = *$ if $u_i \neq w_i$ for $1 \leq i \leq n$.

For example, when u = 0010 and w = 0111, we get H(u,w) = 2 and SQ(u,w) = 0*1*. It is easy to see that SQ(u,w) is the smallest subcube that contains both u and w, and H(u,w) is the dimension of SQ(u,w).

A path in a hypercube is represented by a sequence of nodes in which every two consecutive nodes are physically adjacent to each other in the hypercube. The number of links on a path is called the *length* of the path. An *optimal path* is a path whose length is equal to the Hamming distance between the source and destination nodes. A *shortest path* is a path of the minimal length among all fault-free paths from the source to the destination. Clearly, an optimal path is a shortest path, but a shortest path is not always an optimal path in an injured hypercube. Also, a link of node u is said to be *towards* another node w if the link is in one of the optimal paths between u and w.

Note that due to the special structure of a hypercube, once the source node of a path is given, the path can be described by a *coordinate sequence* that represents the order of the dimensions in which every two consecutive nodes in a path differ [32]. As shown in Figure 4.3, [0001, 0011, 0010, 1010] is an optimal path from the source node 0001 to the destination node 1010, and can also be represented by a coordinate sequence [2, 1, 4]. To facilitate our presentation, we use a superindex R to denote the *reverse* of a coordinate sequence, and the notation $\odot$ to mean an *append* operation. For example, suppose C = [2,1,4] is the coordinate sequence of a path. Then, C^R = [4,1,2] and $C \odot 3$ = [2,1,4,3]. We shall assume that the source and destination nodes are non-faulty.

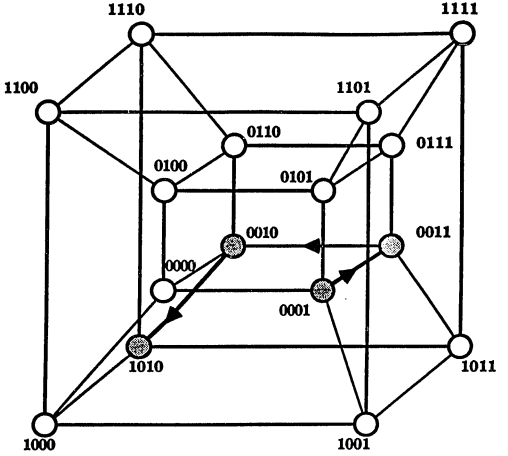


Figure 4.3. An optimal path from 0001 to 1010 .

4.1.2. Routing using depth-first search

We shall develop and analyze an adaptive routing algorithm based on depth-first search, which requires every node to know only the condition of its own links. This algorithm will be shown to successfully route messages between any pair of non-faulty nodes. When the insufficient knowledge on faulty components causes a message to be sent to an intermediate node from which there is no optimal path to the destination node, an alternative path will be chosen in such a way that the connectivity of a hypercube is fully exploited. However, due to the insufficient amount of information on faulty components, the paths chosen by this algorithm may not always be the shortest.

Before describing the routing algorithm, it is necessary to introduce the following proposition which determines relative addresses of those nodes traversed by a given path.

Proposition 4.1: Let $[c_1, c_2, \cdots, c_k]$ be the coordinate sequence of a path in a Q_n starting from node u, and $w_{/u} = w_n w_{n-1} \cdots w_1$ denote the relative address of node w with respect to u. Then, the path specified by $[c_1, c_2, \cdots, c_k]$ ends at w if and only if $\oplus_{i=1}^{k} e^{c_i} = w_{/u}$.

Proof: Traversal of a message along the i-th dimension is the same as inverting the bit in the i-th coordinate of the relative address of its destination. Therefore, traveling along a certain dimension an even number of times has the same effect as not traveling along that dimension at all, and this proposition thus follows. **Q.E.D.**

Let S(C) denote the set of the relative addresses of those nodes reachable by the coordinate sequence $C = [c_1, c_2, \cdots, c_k]$ from a given node. From Proposition 4.1, we obtain $S(C) = \{\oplus_{i=1}^{r} e^{c_i} \mid 1 \le r \le k\}$. For example, a path with the coordinate sequence [2,1,4] from 0000 will traverse nodes S(C) = {0010, 0011, 1011}. We can now describe algorithm A_1 as fol-

lows. To indicate the destination of a message, the coordinate sequence of the remaining path is sent along with the message. In addition, to execute a depth-first search, we shall try to avoid traveling a same node more than once, unless a backtracking is forced. In order to do that those dimensions traveled before will be kept in an order set TD in the sequence they are traveled, and delivered together with the message for the information of intermediate nodes. Note that each intermediate node can determine the addresses of those nodes traveled before from TD^R. Each message is accompanied with an n-bit vector $tag = d_n d_{n-1} \cdots d_1$ which keeps track of "spare dimensions" that are used to choose the dimensions to travel so as to bypass faulty components. All bits in the tag are reset to zero and TD is set to $\varnothing$ when the source node begins routing of a message. Therefore, such a message can be represented as (k, $[c_1, c_2, \cdots, c_k]$, message, TD, tag), where k is the length of the remaining portion of the path and updated as the message travels towards the destination. A message reaches its destination when k becomes zero.

When a node receives a message, it will check the value of k to see if the node is the destination of the message. If not, the node will try to send the message along one of those dimensions specified in the remaining coordinate sequence. Each node will, of course, attempt to route messages via shortest paths first. However, if all the links along those dimensions in the coordinate sequence are either faulty or leading to those nodes traveled before, the node will use the information kept in the tag to choose a spare dimension to route the message via an alternative path. More formally, this routing scheme can be described in an algorithmic form as follows.

Algorithm A_1: Depth-first search routing algorithm.

/* For each node receiving (k, $[c_1, c_2, \cdots, c_k]$, message, TD, tag). */
if k=0 **then** /* the destination is reached */

```
else begin
   for j := 1, k do
     if (the c_j-th link is not faulty) and (e^{c_j} ∉ S(TD^R)) then        ------- *
       begin
         send (k-1,[c_1,...,c_{j-1},c_{j+1},..., c_k], message, TD ⊙c_j, tag) along the c_j-th link;
         stop;  /*terminate Algorithm A_1*/
       end_begin
   end_do

/* If the algorithm is not terminated yet, all dimensions in the coordinate sequence are
blocked because of faulty components.*/
   for j := 1, k do d_{c_j} := 1 end_do; /* record all blocked dimensions in tag.*/

   /* choose a spare dimension */
   if {i : d_i=0, e^i ∉ S(TD^R), 1 ≤ i ≤ n} ≠ ∅ then h := min_{1≤i≤n} {i : d_i=0, e^i ∉ S(TD^R)}
     else if {i : e^i ∉ S(TD^R), 1 ≤ i ≤ n} ≠ ∅ then
       begin
         h := min_{1≤i≤n} {i : e^i ∉ S(TD^R)}
         tag := 0^n;
       end
       else
       begin
         h := max {m | ⊕_{i=1}^{m} e^{TD^R[i]} = 0^n};     /* backtracking */
         tag := 0^n;
       end
   d_h := 1;
   send (k+1, [c_1,c_2..., c_k, h], message, TD ⊙h, tag) along the h-th link;
   stop;  /*terminate Algorithm A_1*/
end_begin
```

Note that an intermediate node can determine whether its i-th link is connected to a node traveled before or not by checking if e^i belongs to $S(TD^R)$. When a backtracking is enforced, the message must be returned to the node from which this message was originally received. That is the reason why we have the equality commented by /* backtracking */ above. It is worth mentioning that instead of keeping the whole path traveled in TD, the search can also be implemented by using a stack. In that case, the operations required for backtracking is

simplified whereas additional provisions will be needed to ensure a same node not to be traveled more than once.

Consider the Q_4 in Figure 4.4, where links 0-01, 1-01 and 100- are faulty. Suppose a message, fm, is routed from u = 0110 to w = 1001. The original message in u = 0110 is (4, [1,2,3,4], fm, Ø, 0000). Following the execution of A_1, node 0110 sends (3, [2,3,4], fm, [1], 0000) to node 0111 which then sends (2, [3,4], fm, [1,2], 0000) to node 0101. Since the 3-*rd* dimensional link of 0101 is faulty, node 0101 will route (1, [3], fm, [1,2,4], 0000) to 1101. However, since the 3-*rd* dimensional link of 1101 is faulty, node 1101 will use the 1-*st* dimension (tag = 0100 then), and send (2, [3,1], fm, [1,2,4,1], 0101) to 1100, which will, in turn, send (1, [1], fm, [1,2,4,1,3], 0101) to 1000. Again, the first link of node 1000 is faulty. The 2-*nd* dimension (tag= 0101 then) will be used and (2, [1,2], fm, [1,2,4,1,3,2], 0111) is routed to 1010. After this, the message will reach the destination 1001 via 1011. The length of the resulting path is 8.

4.1.3. Performance analysis of routing algorithm A_1

From the fact that depth-first search is a search algorithm for general graphs and can start from any node in the graph, we obtain the following lemma.

Lemma 4.1: Under A_1 a message will reach its destination node as long as the source and destination nodes are connected.

Note that the usage of tag, while not affecting the nature of depth-first search, can guide the choice of a spare dimension in order to bypass faulty components. More specifically, we have the theorem below, showing that when the number of faulty components is less than n, the usage of tag can route messages between any two nodes successfully without using TD.

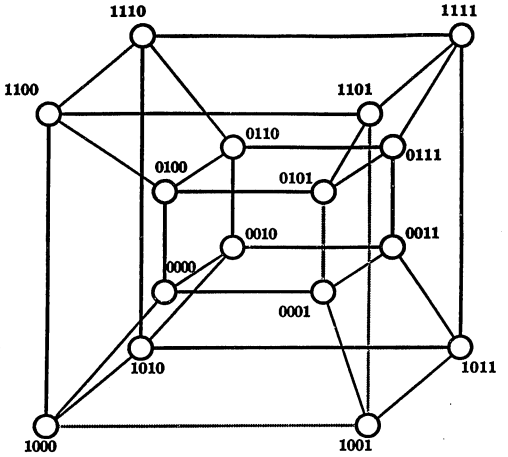


Figure 4.4 . An injured Q_4 where links 0-01, 1-01 and 100- are faulty.

Theorem 4.1: Algorithm A_1 can always route messages from node u to node w within no more than H(u,w) + 2k hops in an injured Q_n without using TD if the number of faulty components, k, is less than n, i.e., k = f+g < n, where f and g are the numbers of faulty links and faulty nodes, respectively.

Proof: Note that each node will try to use a spare dimension only when faulty components are encountered in all the dimensions specified by the coordinate sequence. Those faulty components which block the optimal paths from an intermediate node to the destination node and force the first usage of a spare dimension are called α-*type* blocking components. On the other hand, a faulty component is said to be β-*type* if it is encountered first after using a new spare dimension. For the example routing in Figure 4.4, 1-01 is an α-type blocking component and 100- is a β-type blocking component, whereas the faulty link 0-01 is neither α-type nor β-type. For the example in Figure 4.5 where u = 0000 and w = 1111, 0-11 and -011 are α-type blocking components while 111- is a β-type blocking component. Notice that both the types of blocking components can be either faulty nodes or faulty links. Thus, it is easy to see that the number of both α-type and β-type blocking components in the route determined by A_1 usually increases as the message travels towards its destination.

Let b_h be a β-type blocking component which is encountered first after using a new spare dimension h. We claim that the blocking component b_h does not belong to the set of those blocking components that had already been encountered before. This claim is proved by considering two possible cases of b_h: (i) b_h is a link of the destination node, and (ii) b_h is not a link of the destination node. In the case of (i), b_h is the h-*th* link of the destination node. Since d_h of tag was 0 before the spare dimension h is used, this faulty link had definitely not been encountered before. In the case of (ii), since b_h is the blocking component encountered first after using the spare dimension h, b_h and the set of previous blocking components must

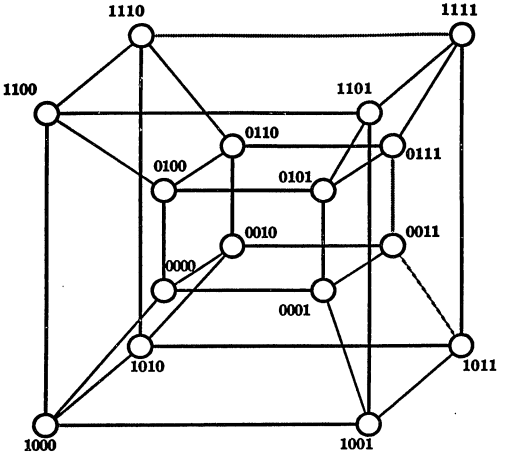


Figure 4.5. An injured Q_4 where links 0-11, -011 and 111- are faulty.

be located in the two different Q_{n-1}'s separated by the dimension h. The claim is thus proved. Since a certain faulty component will not be encountered more than once as long as the number of component failures is less than n, this theorem thus follows. **Q.E.D.**

From the proof of Theorem 4.1, it can be seen that the usage of tag will spread out the spare dimensions chosen and enable the message to avoid running into the faulty components encountered before. In addition, we have the following theorem.

Theorem 4.2: The nodes and links traveled by A_1 form a tree.

Proof: Notice that according to A_1, nodes in TD will not be traveled more than once unless a backtracking is forced, which will travel a link traveled before. This means that those links traveled will not form a cycle and this theorem follows. **Q.E.D.**

Theorem 4.2 leads to the following corollary.

Corollary 4.2.1: The worst case of A_1 needs $H(u,w) + 2(2^n - H(u,w) - 1)$ hops to send a message from u to w.

Proof: When $H(u,w)=k$, there are at most $2^n - k$ nodes where backtracking may occur under A_1. These nodes will form a search tree with $2^n - k$ nodes and $2^n - k - 1$ links. Since every link in the search tree is traveled twice, the message will travel $2(2^n - k - 1)$ hops in the search tree and k hops in its shortest path. **Q.E.D.**

Recall that α-type blocking components are those faulty components forcing the first usage of a spare dimension. To facilitate our presentation, the first node which is forced to use a spare dimension is called an *obstructed node*. For example, the obstructed nodes in the examples of Figures 4.4 and 4.5 are 1101 and 0011, respectively. To illustrate the performance of depth-first search, the model we consider is an injured hypercube with a certain number of faulty components. We assume that all possible distributions of faulty components

are equally likely. Then, we have the following lemma.

Lemma 4.2: Suppose there are f faulty links in a Q_n, and a message is routed by A_1 from node u to node w, where H(u,w) = k. Let h_L be the Hamming distance between the obstructed node and the destination node. Then, $P(h_L=j) \le C_{f-j}^{L-j}/C_f^L$ if $1 \le j \le k$, and $P(h_L=j) = 0$ if $j > k$, where $P(h_L=j)$ is the probability of the case $h_L = j$ and $L = n2^{n-1}$ is the number of links in a Q_n.

Proof: $P(h_L > k) = 0$, since the inequality $h_L > k$ represents an impossible case in which a message is not directed towards its destination before encountering the obstructed node.

Consider the case of $1 \le j \le k$ and assume there are f faulty links in an injured Q_n. Since these faults may occur at any f links in the Q_n, there are C_f^L different configurations (of faulty links) where $L = n2^{n-1}$. Without loss of generality, we can let $u = 0^n$ and $w = 0^{n-k}1^k$. The problem of obtaining $P(h_L=j)$ is then reduced to that of counting the number of configurations which lead to the case of $h_L = j$. We claim that the number of such configurations is less than or equal to C_{f-j}^{L-j}.

When $h_L = j$, the obstructed node must be within the subcube $0^{n-k}*^k$, and all its j links towards w must be faulty (i.e., j α-type blocking components). Although there are many possible locations of the obstructed node, according to the systematic procedure of A_1, the location of the obstructed node is determined by those non-α-type faulty links which are not within $0^{n-k}*^k$. Suppose $x = 0^{n-k+j}1^{k-j}$ is the obstructed node, then the j links of node x within SQ(x,w) are faulty and there are C_{f-j}^{L-j} different distributions of these non-α-type faulty links. When these non-α-type faulty links cause node y, instead of x, to be the obstructed node, we exchange the links (including faulty links) in SQ(y,w) with those in SQ(x,w), and obtain a configuration which leads to the case when the obstructed node is y and $h_L = j$. Notice that some of the C_{f-j}^{L-j} different distributions of non-α-type faulty links may lead to $h_L > j$, meaning

that the number of configurations leading to $h_L = j$ is less than or equal to C_{F-j}^{L-j}. This lemma thus follows. **Q.E.D.**

Notice that as pointed out in [34], the number of faulty nodes required for a coordinate sequence C to be a resulting path by depth-first search is equal to the number of inversions in C [50]. Furthermore, it is shown in [34] that the probability for a message to be routed in an optimal path to a destination node n hops away in the presence of g faulty nodes in a Q_n is

$$\begin{bmatrix} 2^n - 2 \\ g \end{bmatrix}^{-1} \sum_{k=0}^{n(n-1)/2} I_n(k) \begin{bmatrix} 2^n - 2 - (n-1) - k \\ g - k \end{bmatrix},$$

where $I_n(k)$ denotes the number of permutations on n numbers with exactly k inversions. The value of $I_n(k)$ can be obtained from its generation function $G_n(z) = \sum_{j=0}^{n(n-1)/2} I_n(j) z^j =$ $(1+z)(1+z+z^2) \cdots (1+z+z^2+\cdots+z^{n-1})$ by $I_n(k) = \frac{1}{k!} \frac{d^k G_n(0)}{dz^k}$. Following the same concept, the probability for a message to be routed in an optimal path to a destination node n hops away in the presence of f faulty links in a Q_n can be expressed as

$$\begin{bmatrix} n2^{n-1} \\ f \end{bmatrix}^{-1} \sum_{k=0}^{n(n-1)/2} I_n(k) \begin{bmatrix} n2^{n-1} - n - k \\ f - k \end{bmatrix}.$$

It is worth mentioning that in light of Lemma 4.2, a lower bound of the probability for a message to be routed in an optimal path in the presence of f faulty links in a Q_n can be derived in a closed form expression as shown in the following theorem, which shows that Algorithm A_1 can route a message to the destination via an optimal path with a very high probability.

Theorem 4.3: Suppose there are f faulty links in a Q_n. Algorithm A_1 will route a message from a node u to to another node w via an optimal path between u and w with a proba-

bility greater than $1 - \sum_{j=1}^{k} \frac{C_{L-j}^{L-j}}{C_f^L}$, where $L = n2^{n-1}$, and $H(u,w) = k$.

Proof: From Lemma 4.2, the probability that A_1 has to use spare dimensions is $\sum_{j=1}^{k} P(h_L = j) \le \sum_{j=1}^{k} \frac{C_{L-j}^{L-j}}{C_f^L}$. Thus, the probability that A_1 will not use any spare dimension at all is $1 - \sum_{j=1}^{k} P(h_L = j) \ge 1 - \sum_{j=1}^{k} \frac{C_{L-j}^{L-j}}{C_f^L}$. **Q.E.D.**

When there are n–1 faulty links in a Q_n, the lower bound of the probability that A_1 will result in the optimal path routing can be derived as follows.

Corollary 4.3.1: Suppose there are n–1 faulty links in a Q_n. Algorithm A_1 will route a message from a node u to another node w via an optimal path between u and w with a probability greater than $1 - \frac{r_1(1 - r_1^k)}{(1-r_1)}$, where $H(u,w) = k$ and $r_1 = \frac{n-1}{n2^{n-1}}$.

Proof: From Theorem 4.3, we have

$$\sum_{j=1}^{k} \frac{C_{n-1-j}^{L-j}}{C_{n-1}^{L}} = \frac{n-1}{L} + \frac{(n-1)(n-2)}{L(L-1)} + \cdots + \frac{(n-1)(n-2)\cdots(n-k)}{L(L-1)\cdots(L-k+1)}, \text{ where } L = n2^{n-1}.$$

Since $r_1 = \frac{n-1}{L} > \frac{n-2}{L-1} > \cdots \frac{n-k}{L-k+1}$, we get

$$\sum_{j=1}^{k} \frac{C_{n-1-j}^{L-j}}{C_{n-1}^{L}} < r_1 + r_1^2 + \cdots + r_1^k = \frac{r_1(1 - r_1^k)}{(1-r_1)}, \text{ implying } 1 - \sum_{j=1}^{k} \frac{C_{n-1-j}^{L-j}}{C_{n-1}^{L}} > 1 - \frac{r_1(1 - r_1^k)}{(1-r_1)}.$$

This corollary thus follows. **Q.E.D.**

It can be seen from Theorem 4.3 that A_1 will route a message to its destination via an optimal path with a rather high probability in the presence of faulty links. Similarly to the case of faulty links, the performance of A_1 can be analyzed in terms of *node* failures as

described by Lemma 4.3 below whose proof is similar to that of Lemma 4.2 and thus omitted.

Lemma 4.3: Suppose there are g faulty nodes in a Q_n, and messages are to be routed from an arbitrary node u to another node w, where H(u,w) = k. Let h_N be the Hamming distance between the obstructed node and the destination node w. Then, $P(h_N=j) \le \frac{C_{g-j}^{N-3-j}}{C_g^{N-2}}$ if $2 \le j \le k$, and $P(h_N = j) = 0$ if $j = 1$ or $j > k$, where $N = 2^n$ is the total number of nodes in a Q_n.

As mentioned before, the probability for a message to be routed in an optimal path to a destination node n hops away in the presence of g faulty nodes in a Q_n is

$$\begin{Bmatrix} 2^n - 2 \\ g \end{Bmatrix}^{-1} \sum_{k=0}^{n(n-1)/2} I_n(k) \begin{Bmatrix} 2^n - 2 - (n-1) - k \\ g - k \end{Bmatrix}.$$

Also, from Lemma 4.3 and the reasoning in the proof of Theorem 4.3, we can obtain a lower bound for the corresponding probability in Theorem 4.4 and its corollary, which shows that A_1 can also route a message between any pair of non-faulty nodes in an injured hypercube via an optimal path with a high probability.

Theorem 4.4: Suppose there are g faulty nodes in a Q_n. Algorithm A_1 will route a message from a node u to another node w via an optimal path between u and w with a probability greater than $1 - \sum_{j=2}^{k} \frac{C_{g-j}^{N-3-j}}{C_g^{N-2}}$, where H(u,w) = k.

Corollary 4.4.1: Suppose there are n−1 faulty nodes in a Q_n. Algorithm A_1 will route a message from a node u to another node w via an optimal path between u and w with a probability greater than $1 - \frac{(n-1)r_2(1-r_2^{k-1})}{(2^n-2)(1-r_2)}$, where H(u,w) = k, and $r_2 = \frac{n-2}{2^n-3}$.

Proof: Notice that

$$\frac{C_{n-1-j}^{N-3-j}}{C_{n-1}^{N-2}} = \frac{(n-1)(n-2)\cdots(n-j+1)(n-j)(N-n-1)}{(N-2)(N-3)\cdots(N-j-2)}$$

$$\leq \frac{(n-1)(n-2)\cdots(n-j+1)(n-j)}{(N-2)(N-3)\cdots(N-j-1)} \quad \text{(Since } N-n-1 \leq N-j-2.\text{)}$$

$$= \frac{n-1}{N-2} r_2^{j-1}.$$

By letting $g = n-1$ in Theorem 4.4 we get $1 - \sum_{j=2}^{k} \frac{C_{n-1-j}^{N-3-j}}{C_{n-1}^{N-2}} > 1 - \sum_{j=2}^{k} \frac{n-1}{N-2} r_2^{j-1} =$

$1 - \frac{n-1}{N-2} \sum_{j=2}^{k} r_2^{j-1} = 1 - \frac{(n-1)r_2(1-r_2^{k-1})}{(N-2)(1-r_2)}$, where $N = 2^n$. **Q.E.D.**

Furthermore, as it will be shown below, the expected length of a path resulting from the use of A_1 is very close to that of an optimal path, i.e., the Hamming distance between the source and destination nodes. Before analyzing the quality of the paths selected by A_1, it is necessary to introduce the following proposition.

Proposition 4.2: Let $\{p_i\}_{i=1}^{n}$ and $\{q_i\}_{i=1}^{n}$ be, respectively, two decreasing sequences with $p_n = q_n = 0$. Suppose $p_i \leq q_i$ for $1 \leq i \leq n-1$, then $\sum_{i=1}^{n-1} i(p_i - p_{i+1}) \leq \sum_{i=1}^{n-1} i(q_i - q_{i+1})$.

Proof: Let $d_i = q_i - p_i$ for $1 \leq i \leq n$. Then, we get

$$\sum_{i=1}^{n-1} i(q_i - q_{i+1}) - \sum_{i=1}^{n-1} i(p_i - p_{i+1}) = \sum_{i=1}^{n-1} i(d_i - d_{i+1})$$

$$= \sum_{i=1}^{n-1} d_i \geq 0. \quad \text{(Since } d_i \geq 0 \text{ for } 1 \leq i \leq n-1.\text{)} \quad \textbf{Q.E.D.}$$

We can now derive the following important theorem.

Theorem 4.5: Let u and w be a pair of nodes with $H(u,w) = n$ in an injured Q_n which contains $n-1$ faulty links. Let H_1 be the length of a path between u and w that is chosen by

A_1. Then, $E(\Delta H_1) \le \frac{n-1}{2^{n-2}}$, where E(x) denotes the expected value of a random variable x, and $\Delta H_1 = H_1 - n$.

Proof: Notice that $P(\Delta H_1 \ge 2i) \le \sum_{j=1}^{n-i} P(h_L = j)$. Then,

$$E(\Delta H_1) = \sum_{i=1}^{n-1} 2i\, P(\Delta H_1 = 2i)$$

$$= \sum_{i=1}^{n-1} 2i \left[P(\Delta H_1 \ge 2i) - P(\Delta H_1 \ge 2(i+1)) \right]$$

$$\le \sum_{i=1}^{n-1} 2i \left[\sum_{j=1}^{n-i} P(h_L = j) - \sum_{j=1}^{n-i-1} P(h_L = j) \right] \quad \text{(By Proposition 4.2.)}$$

$$= \sum_{i=1}^{n-1} 2i\, P(h_L = n-i)$$

$$\le \sum_{i=1}^{n-1} 2(n-i) \frac{C_{n-1-i}^{L-i}}{C_{n-1}^{L}} \quad \text{(By Lemma 4.1 and } L = n2^{n-1}.\text{)}$$

$$< \sum_{i=1}^{n-1} 2(n-i) r_1^i \quad (r_1 = \frac{n-1}{L})$$

$$= 2n \sum_{i=1}^{n-1} r_1^i - 2 \sum_{i=1}^{n-1} i r_1^i$$

$$= 2nr_1 + 2nr_1(r_1 + r_1^2 + \cdots + r_1^{n-2}) - 2(r_1 + 2r_1^2 + \cdots + (n-1)r_1^{n-1})$$

$$< 2nr_1 = \frac{n-1}{2^{n-2}}. \quad \text{(Since } nr_1 < 1.\text{)} \quad \textbf{Q.E.D.}$$

Corollary 4.5.1: Suppose messages are to be routed from an arbitrary node u to another node w in an injured Q_n which contains n–1 faulty *nodes*. Let H_2 be the length of a path between u and w that is chosen by A_1, and let $\Delta H_2 = H_2 - n$, where H(u,w) = n. Then, $E(\Delta H_2) \le \frac{2n(n-1)(n-2)}{(2^n-2)(2^n-3)}$.

Proof: Since $P(\Delta H_2 \geq 2i) \leq \sum_{j=2}^{n-i} P(h_N = j)$, using the same reasoning as in the proof of Theorem 4.5, we get

$$E(\Delta H_2) = \sum_{i=1}^{n-2} 2i\ P(\Delta H_2 = 2i) = \sum_{i=1}^{n-2} 2i \left[P(\Delta H_2 \geq 2i) - P(\Delta H_2 \geq 2(i+1)) \right]$$

$$\leq \sum_{i=1}^{n-2} 2i \left[\sum_{j=2}^{n-i} P(h_N = j) - \sum_{j=2}^{n-i-1} P(h_N = j) \right] \quad \text{(by Proposition 4.2.)}$$

$$= \sum_{i=1}^{n-2} 2i\ P(h_N = n-i)$$

$$\leq \sum_{i=2}^{n-1} 2(n-i) \frac{(n-1)r_2^{i-1}}{(N-2)} \quad \text{(where } N = 2^n \text{ and } r_2 = \frac{n-2}{N-3}\text{)}$$

$$= \frac{2(n-1)}{N-2} \sum_{i=2}^{n-1} (n-i) r_2^{i-1}$$

$$= \frac{2(n-1)}{N-2} \left[nr_2(1 + r_2 + r_2^2 + \cdots + r_2^{n-3}) - (2r_2 + 3r_2^2 + \cdots + (n-1)r_2^{n-2}) \right]$$

$$= \frac{2(n-1)}{N-2} \left[nr_2 + nr_2(r_2 + r_2^2 + \cdots + r_2^{n-3}) - (2r_2 + 3r_2^2 + \cdots + (n-1)r_2^{n-2}) \right]$$

$$\leq \frac{2(n-1)nr_2}{N-2} = \frac{2n(n-1)(n-2)}{(2^n-2)(2^n-3)}. \quad \text{(Since } nr_2 < 1.\text{)} \quad \textbf{Q.E.D.}$$

As shown above, Algorithm A_1 routes messages via optimal paths with a high probability, and the expected length of a resulting path is very close to the Hamming distance between the source and destination nodes. However, due to the absence of information at each node on components other than its own links, the presence of a certain faulty component is not known until a message gets to the faulty component. This may force an intermediate node to use a spare dimension for routing messages around the faulty component, thus increasing the length of the actual path taken.

4.1.4. Extension: Propagating network information to neighboring nodes

In order to route messages more efficiently, each node needs to be equipped with more information than that on its own communication links. Consider a routing scheme which is a modified version of Algorithm A_1. In addition to keeping track of the condition of its own links only, every node also makes this information available to all its neighboring nodes. Thus, every node will become aware of not only the condition of its own links but also that of those links which are one hop away from it. To use this information, the conditional statement in A_1 marked with * is modified in such a way that every intermediate node checks one more hop in the coordinate sequence of each message, and uses a spare dimension to bypass faulty components if necessary. Clearly, due to the nature of depth-first search, this modified routing scheme can also successfully route a message to any other node. More specifically, we have the following lemma for the performance of this routing scheme.

Lemma 4.4: Suppose there are f faulty links in an injured Q_n, and a message is to be routed from node u to node w where H(u,w) = k. Let h_C be the Hamming distance between the obstructed node and the destination when each node is informed about the condition of its neighboring nodes. Then, $P(h_C=j) \le \sum_{i=1}^{j} C_i^j C_{f-j-j+2i}^{L-ij-j+2i} / C_f^L$ if $2 \le j \le \min\{1+\lceil \frac{f}{2} \rceil,\ k-1\}$, $P(h_C=k) \le \sum_{i=0}^{k} C_i^k C_{f-ik-k+2i}^{L-ik-k+2i} / C_f^L$ and $P(h_C=j) = 0$ otherwise, where $L = n2^{n-1}$.

Proof: Let x denote the obstructed node. First, consider the case when the obstructed node is the source node, i.e., $h_C = k$. Clearly, x has k links within SQ(x,w). There will be no optimal path from x to w if and only if each of these k links is either faulty or connected to a node with k–1 faulty links towards w. Suppose x has exactly i non-faulty links towards w. Then, there are C_i^k ways to determine which of x's links are non-faulty. For each case, there are (k–i) + i(k–1) specific faulty links, thus resulting in $C_{f-ik-k+2i}^{L-ik-k+2i}$ different configurations of

faulty links. The expression for the case of $P(h_C = k)$ thus follows.

Since every node is informed about the condition of the links one hop away, a message will not be routed to any node whose every link towards w is faulty. Thus, every link of the obstructed node towards w can be faulty only when the obstructed node is the source node. This is the reason why two different expressions are needed for the two cases: $2 \le h_C \le k-1$ and $h_C = k$. Notice, however, that when $h_C = j \ne k$ and the obstructed node x has i non-faulty links towards w, these i links are connected to those nodes with j–1 faulty links towards w. Therefore, from the fact that the total number of faulty links $i(j-1) + j - 1$ is less than f, we get $j \le 1 + \lceil \frac{f}{2} \rceil$, and thus, this lemma follows. **Q.E.D.**

To illustrate the improvement of routing efficiency with the above additional information at each node, let $p_L(j) = C^{L-j}_{f-j}/C^L_f$ and $p_C(j) = \sum_{i=1}^{j} C^i_f C^{L-ij-j+2i}_{f-ij-j+2i}/C^L_f$. As shown in Lemma 4.2 and Lemma 4.4, $P(h_L=j) \le p_L(j)$ and $P(h_C=j) \le p_C(j)$. It can be verified that $p_C(j) < p_L(j)$ for $j > 2$, meaning that $P(h_C=j)$ has a smaller upper bound than $P(h_L=j)$. Note, however, that $h_C(2) > h_A(2)$. This is based on the fact that, under our modified routing scheme, an intermediate node located two hops away from the destination may foresee the unreachability from itself to the destination and use spare dimensions to bypass those faulty components which could not be seen by the same intermediate node under A_1. The upper bound of $P(h_C = 2)$ is thus greater than that of $P(h_L = 2)$. (Note, however, that $h_C(2) < h_A(2) + h_A(1)$.) Therefore, in light of the reasoning in Theorem 4.5, routing efficiency is improved with the additional information at each node.

Clearly, routing efficiency can be improved further by propagating the information on faulty components more than one hop. Notice, however, that the above modified scheme still cannot guarantee the shortest path routing. When two certain nodes communicate with each

other frequently, the practicality of using depth-first search approach needs further justification, since extra hops will be traveled during each transmission. Although each node can always find a shortest fault-free path from itself to any other node if it contains the information on every faulty component, it is impractical to maintain and update such information, especially when the size of the network is very large. Consequently, it is important to determine the information required for the shortest path routing.

4.2. Routing with Global Information

To route the message via the shortest path, we develop routing schemes using global information. By global information, we mean the knowledge on the network condition, in addition to those on its own links. A routing scheme by propagating faulty components will be introduced in Section 4.2.1, which, however, is applicable only for an injured Q_n with less than n faulty components. Then, a routing scheme by exchanging minimal delay vectors will be presented in Section 4.2.2, which is able to route messages via the shortest paths in an injured hypercube of an arbitrary number of faulty components. The amount of information essential for the shortest path routing will also be investigated.

4.2.1. Routing by propagating information on faulty components

To reduce the amount of information at each node required for the shortest path routing, the unnecessary propagation of information on faulty components should be avoided. Notice that a faulty node can be viewed as a node with its all links faulty. Therefore, the network information kept at each node can be represented by a set of addresses of faulty links. As will be proved later, when the number of faulty components is less than n in a Q_n, each node does not have to propagate the information on faulty links to its neighboring nodes unless

these faulty links block all the optimal paths from itself to another node. In other words, only when node u finds that all its optimal paths to another node, say x, have been blocked by a set of faulty links F, node u will propagate the information on F to its neighboring nodes so as to prevent them from choosing node u as a next hop toward node x.

Note that the coordinate sequence of an optimal path from u to w consists of H(u,w) different numbers representing those dimensions in which u and w differ, meaning that there are H(u,w)! different optimal paths from u to w. Then, we have the following proposition, describing the effect of a link failure on the optimal paths between the two nodes.

Proposition 4.3: Let N(u\ w, r) be the number of optimal paths from u to w which traverse a link r. Then,

(i). For any link $r \in SQ(u,w)$, $N(u \setminus w, r) = [H(u,w)-H(u,x)-1]!H(u,x)!$, where x is the one of the link r's two end nodes that is closer to u.

(ii). For any link $r \notin SQ(u,w)$, $N(u \setminus w, r) = 0$.

For example, if u = 0100, w = 1001 and r = 0-01, then x = 0101 and N(u\ w, r) = 2!1! = 2. On the other hand, if r = 0-11, then N(u\ w, r) = 0, since 0-11 $\notin$ SQ(u,w) = **0*. In light of Proposition 4.3, we can derive the condition for a set of faulty links to block all the optimal paths between any given pair of nodes as follows. Let r_x and r_y be the relative addresses of any two links with respect to node u. Then, the link r_y is called a *downstream* link of r_x, written as $r_x < r_y$, if r_x appears before r_y in an optimal path from u to r_y^-. Note that $r_x = x_n x_{n-1} \cdots x_1 < r_y = y_n y_{n-1} \cdots y_1$ iff (1) $x_i \leq y_i$ for $1 \leq i \leq n$, where the symbol '-' is ordered such that $0 \leq - \leq 1$, and (2) r_x and r_y are the links placed in different dimensions. This fact results in a straightforward procedure for determining if a link is a downstream link of another. Each node can thus store the relative addresses of faulty links as a partially

ordered set [71]. A set of relative addresses of faulty links is called a *linear set* if for any two links r_x and r_y in the set, either $r_x < r_y$ or $r_y < r_x$. It can be verified that an optimal path from u to w will traverse all links in a set of links B only if B is a linear set. Let M(u\ w, F) be the number of optimal paths from u to w which traverse *every* link in F. Then, following the concept of exclusion and inclusion [54], we obtain the following lemma which can determine the number of optimal paths blocked by a set of faulty links.

Lemma 4.5: Given a set of faulty links F, the number of the optimal paths from u to w blocked by the links in F is $N(u\backslash w, F) = \sum_{i=1}^{r}(-1)^{i+1}m_i$, where $m_i = \sum_{B_i \in F \cap SQ(u,w)} M(u\backslash w, B_i)$ and B_i denotes a linear set of i links.

Clearly, the condition for a set of faulty links to block all the optimal paths between two nodes u and w is $N(u\backslash w, F) = H(u,w)!$. The operations of Algorithm A_2 can be outlined as follows. Each node keeps the information about two types of faulty links in the form of relative addresses. The first type, denoted by F_0, is the set of those faulty links whose status has not yet been propagated to neighboring nodes, whereas the second type, denoted by F_1, is the set of those faulty links whose status has already been propagated to neighboring nodes.

Since relative addresses of faulty links of a node are kept in that node, the information on F_0 must be modified in accordance with the addresses of receiving nodes when it is propagated to neighboring nodes. A formal description of the algorithm for the determination and modification of link failure information is given below.

Algorithm A_2: Collection of failure information for the shortest path routing.

Testing

/* Each node tests all its communication links.*/

if (the k-th link of the node is faulty) **then**
begin

```
        F1 := F1 ∪ {e^k};
        for i := 1 to n do
          if i ≠ k then send e^k ⊕ e^i along the i-th dimension;
        Propagation;
      end
```

Receiving

```
/* For each node receiving the information on the failure of the link r.*/
      if r ∉ F then          /*F = F0 ∪ F1 */
        begin
          F0 := F0 ∪ {r};
          Propagation;
        end
```

Propagation

```
    if N(0^n\ r^-, F) = H(0^n,r^-)! then
      begin
        F0 := F0 − {r};
        F1 := F1 ∪ {r};
        /* Propagate the information on the failure of r to neighboring nodes.*/
        for i := 1 to n do send r ⊕ e^i along the i-th dimension;
        /* Check if propagation of information on other faulty links is necessary.*/
        for all r_x ∈ F0 and r ∈ SQ(0^n, r_x^-) do
          if N(0^n\ r_x^-, F) = H(0^n, r_x^-)! then
            begin
              /* Propagate the information on the failure of r to neighboring nodes.*/
              for i := 1 to n do send r_x⊕e^i along the i-th dimension;
                F0 := F0 − {r_x};
                F1 := F1 ∪ {r_x};
            end
    end
```

When a node receives the information on the failure of link r, it will update its F_0 and F_1 accordingly, and check if any further propagation of information on other link failures in F_0 is required. For example, consider the Q_4 in Figure 4.4. By executing A_2, each node can determine $F = F_0 \cup F_1$ as well as the information to be propagated to neighboring nodes, F_1. Table 4.1 shows the information to be kept in each node. Notice that the faulty links are

nodes	F_0	F_1
0000	{0-01,1-01,100-}	∅
0001	{100-}	{0-00,1-00}
0010	∅	∅
0011	{0-10,1-10}	∅
0100	{0-01,1-01,110-}	∅
0101	∅	{0-00,1-00}
0110	∅	∅
0111	{0-10,1-10}	∅
1000	{0-01}	{000-}
1001	∅	{1-00,0-00,000-}
1010	{001-}	∅
1011	{1-10,0-10,001-}	∅
1100	{1-01}	{0-01,010-}
1101	{010-}	{1-00,0-00}
1110	{0-11,011-}	∅
1111	{1-10,0-10}	∅

Table 4.1. Information in each node generated by Algorithm A_2 for the injured hypercube in Figure 4.4.

represented in each node by their relative addresses with respect to the node. To see the fact that A_2 can achieve the shortest path routing when the number of faulty components is less than n, we need the following theorem, which was introduced in [73].

Theorem of connectivity [73]: Let u and w be two arbitrary nodes in a Q_n such that $H(u,w) = k$. Then, there are exactly n disjoint paths of length less than or equal to k+2 from u to w. These paths are composed of k disjoint paths of length k, and $(n - k)$ disjoint paths of length k+2.

Then, we have the theorem below.

Theorem 4.6: Under Algorithm A_2, every node can obtain the failure information essential for the shortest path routing as long as the number of faulty components is less than n.

Proof: Notice that the necessary and sufficient condition for all the optimal paths from node u to node w to be blocked is that "for all $z \in SQ(u,w)$ reachable from u via an optimal path, then there is no optimal path from z to w." Since every node propagates the corresponding failure information to its neighboring nodes if all its optimal paths to a certain node are blocked, the fact that every node will know if all its optimal paths to a certain node are blocked can be proved by induction.

When node u finds all its optimal paths to w are blocked, there are at least $H(u,w) = k$ faulty components in SQ(u,w). Note that there are still $n - k$ disjoint paths of length k+2 via the neighboring nodes of u which are not within SQ(u,w), and at least one of them is fault-free because there are at most $n - 1$ faulty components in the Q_n. Since those neighboring nodes not having any optimal path to w will propagate the corresponding failure information to u to prevent u from choosing one of them as a next hop, this theorem follows. **Q.E.D.**

Theorem of connectivity and Theorem 4.6 lead to the following corollary.

Corollary 4.7.1: Algorithm A_2 can route a message from node u to node w in H(u,w)+2 hops as long as the number of faulty components is less than n.

When a node needs to send a message to another node, it will use its information on faulty components to determine the coordinate sequence of a shortest fault-free path to the destination node as if it had the information on every faulty component in the entire network. Then, according to the first entry of the coordinate sequence, the source node will determine the next hop of the message and its associated coordinate sequence. On the other hand, when a node receives a message and the coordinate sequence of the remaining part of the path, it will check whether the remaining path contains faulty links and permute the order of entries in the coordinate sequence to bypass the faulty components, if necessary.

For example, consider the injured Q_4 in Figure 4.4 where node 0110 is trying to send a message to node 1001. The source node is not aware of any faulty link, and thus, routes a message (3, [2,3,4], fm) to 0111. However, as indicated in Table 4.1, the node 0111 will find the path [2,3,4] is faulty, since the path will encounter the faulty link 0-01 whose relative address with respect to 0111 is $0\text{-}01_{/0111}$ =0-10. Thus, a new non-faulty path [3,2,4] is determined by 0111. The message will be routed to 0011, and then to the destination 1001 via 1011. The length of the resulting path is 4. This is much less than the length of the path determined by A_1, 8.

Notice, however, when the number of faulty components is more than n in a Q_n, each node may not be able to gather enough information required for the shortest path routing, since too many component failures may block the propagation of failure information. For example, consider the injured hypercube in Figure 4.6. The link failures known to node 1000 are 000-, 001-, -100 and 100-, which will provide node 1000 the knowledge that there is no

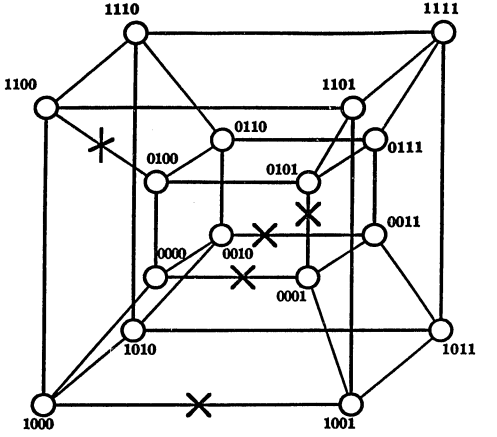


Figure 4.6. An injured hypercube where node 1000 does not have enough information for the shortest path routing.

optimal path to 0001. However, without the knowledge on the failure of link 0-01, node 1000 may choose [1000, 0000, 0100, 0101, 0001] as its shortest path to node 0001, instead of choosing a fault-free path [1000, 1011, 1001, 0001].

4.2.2. Routing by exchanging minimal delay vectors

As shown above, A_2 is not sufficient to guarantee the shortest path routing when the number of faulty components is more than n in a Q_n. To remedy this, we shall use the idea developed in Chapter 2 to achieve the shortest path routing. Recall that under APRS the path from one node to every other node is not determined in advance. Instead, every node maintains a network delay table to record the shortest delay via each link emanating from the node. When a node is to send a message to another node, it will check its network delay table and determine the next hop of the message for the shortest path routing. A *minimal delay vector* in a node, which contains the delays of the shortest paths from that node to all other nodes, is passed to all of its adjacent nodes as routing information. After receiving minimal delay vectors from adjacent nodes, every node will update its network delay table accordingly.

More formally, the operations for maintaining and updating the network delay vector for each hypercube node can be described as follows. Let the entry at row i and column w of the network delay table of node u be denoted by $d_{u\backslash i,w}$, which represents the delay of the shortest path from node u via dimension i to node w. Also, denote the minimal delay from u to w as $d_{u,w}$, i.e., $d_{u,w} = \min_{1 \le i \le n} \{d_{u\backslash i,w}\}$. Then, after receiving $d_{v,w}$ from node $v = u \oplus e^i$, node u will calculate $d_{u\backslash i,w} = d_{v,w} + 1$ and update $d_{u,w}$ if necessary. The new $d_{u,w}$ will, in turn, be sent to adjacent nodes $u \oplus e^i$, $1 \le i \le n$ as routing information. For example, the network delay tables for nodes 000, 100 and 101 in Figure 4.7 are given in Table 4.2a, 4.2b and 4.2c, respectively. The routing information generated by node 1001 is also shown in Figure 4.2d.

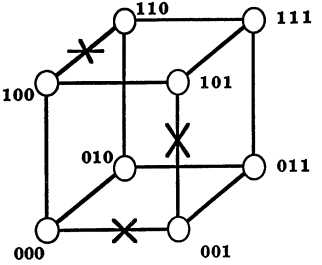


Figure 4.7. An example Q_3 for the routing scheme based on the minimal delay tables.

destination / dimension	001 *	010	011	100	101	110 *	111
1 (001)	∞	∞	∞	∞	∞	∞	∞
2 (010)	3	1	2	3	4	2	3
3 (100)	5	3	4	1	2	4	3

(a). Network delay table of node 000.

destination / dimension	000	001 *	010	011	101	110 *	111
1 (101)	3	4	4	3	1	3	2
2 (110)	∞	∞	∞	∞	∞	∞	∞
3 (000)	1	4	2	3	3	3	4

(b). Network delay table of node 100.

destination / dimension	000	001 *	010	011	100	110 *	111
1 (100)	2	5	3	4	1	4	3
2 (111)	4	3	3	2	3	2	1
3 (110)	∞	∞	∞	∞	∞	∞	∞

(c). Network delay table of node 101.

000	001 *	010	011	101	110	111
1	4	2	3	1	3	2

(d). Routing information generated by node 100.

Table 4.2. Network delay tables for some hypercube nodes in Figure 4.7.

It can be observed that in Table 4.2 several entries contain the information which can be directly determined by the regularity of the hypercube topology. More specifically, if the entry $d_{u\backslash i,w}$ contains the value $H(u \oplus e^i,w) + 1$, this entry is redundant since the information it contains is implied by the hypercube topology. In light of the above fact, the operations for generating the network delay table are modified as follows. In the network delay table of node u, a column associated with a destination node w is generated only when either of the following two cases occurs.

1. Node u receives routing information $d_{v,w}$ from its neighboring node v.
2. Node w is within the minimal subcube which contains node u and its faulty links.

Then, each node, instead of exchanging routing messages with neighboring nodes periodically, propagates routing messages to its neighboring nodes only when its network delay table is updated and a new minimal delay to some other node results. For example, according to the above modification, only those columns whose destination nodes are marked by *'s are required to be kept in Table 4.2. It can be verified that the remaining network information will still be sufficient for the shortest path routing. It is worth mentioning that, for the network in Figure 4.6, we can obtain the network delay tables as shown in Table 4.3 from which the shortest path from 1000 to 0001 is determined. In addition, we have the following lemma for this routing scheme.

Lemma 4.6: In the network delay table of node u under the 1-st order routing strategy, $|d_{u\backslash i,w} - d_{u\backslash j,w}| \leq 2$, for $1 \leq i, j \leq n$.

It can be seen that looping may occur in the presence of component failures under this routing scheme, which is equivalent to the 0-th order routing strategy in Chapter 2. Clearly, the approach of using high order routing strategies can be applied to eliminate the looping. For example, the network delay table under the 1-st order routing strategy for the injured

destination / dimension	0001	0011	1001 †	1100
1 (0001)	∞	∞	∞	∞
2 (0010)	5	4	4	4
3 (0100)	5	4	4	4
4 (1000)	3	4	2	2

(a). Network delay table of node 0000.

destination / dimension	0001	0100	0011	1001 †
1 (1001)	2	4	3	1
2 (1010)	4	4	3	3
3 (1100)	4	4	5	3
4 (0000)	4	2	5	3

(b). Network delay table of node 1000.

destination / dimension	0001 †	0100	1001 †
1 (1101)	3	3	2
2 (1110)	5	3	4
3 (1000)	3	3	2
4 (0100)	∞	∞	∞

(c). Network delay table of node 1100.

Table 4.3. Network delay tables for some hypercube nodes in Figure 4.6.

hypercube in Figure 4.7 is shown in Table 4.4.

From the hypercube topology, we have the following theorem.

Theorem 4.7: Every loop in an injured Q_n contains even number of hops.

Proof: From Proposition 4.1, we know every cycle, starting and ending at the same node will traverse every dimension even number of times. This theorem thus follows. **Q.E.D.**

Then, we have the following corollary.

Corollary 4.7.1: The required order of routing strategy for an injured hypercube is odd.

It can be seen that the information about an isolated faulty link needs to be propagated only to its neighboring nodes, whereas the information about clustered faulty links has to be propagated a little farther to ensure each message to be routed via a shortest path. For example, the network delay table of node 100 in Figure 4.7 is affected by the failure of links -01 and 00- (two hops away), while the failure of link 1-0 only affects the network delay table of those nodes one hop away. This agrees well with our intuition, since clustered faulty components are likely to block more optimal paths between a pair of nodes, and thus, have to be kept by those nodes far away from them to achieve the shortest path routing. Clearly, when the size of the hypercube increases, the zone of nodes whose network delay tables are influenced by a faulty component will become rather small relative to the size of the entire network.

destination / dimension	001 *	010 *	011	100 *	101	110 *	111
1 (001)	∞	∞	∞	∞	∞	∞	∞
2 (010)	3	1	2	5	4	2	3
3 (100)	5	5	4	1	2	4	3

(a). Network delay table of node 000.

destination / dimension	000 *	001 *	010	011	101 *	110 *	111
1 (101)	5	4	4	3	1	3	2
2 (110)	∞	∞	∞	∞	∞	∞	∞
3 (000)	1	4	2	3	5	3	4

(b). Network delay table of node 100.

destination / dimension	000	001 *	010	011	100 *	110 *	111 *
1 (100)	2	5	3	4	1	4	5
2 (111)	4	3	3	2	5	2	1
3 (110)	∞	∞	∞	∞	∞	∞	∞

(c). Network delay table of node 101.

001 *	010 *	011	101	110	111
4	4	3	1	3	2

(d). Routing information generated by node 100 for node 000.

000	001 *	010 *	011	110	111
1	4	2	3	3	4

(e). Routing information generated by node 100 for node 101.

Table 4.4. Network delay tables for some hypercube nodes in Figure 4.7.

CHAPTER 5

TASK ALLOCATION IN HETEROGENEOUS MULTICOMPUTER SYSTEMS

In a multicomputer system, a task is usually decomposed into a set of cooperating task modules which are then assigned to a set of processors in order to exploit the inherent parallelism in the task execution [18, 19, 76, 77]. Each task can thus be described by an undirected graph called the *task graph*, $G_T = (V_T, E_T)$, where V_T is the set of nodes, each representing a task module, and $E_T \subseteq V_T \times V_T$ is the set of edges, each representing the communication required between the two task modules connected by the edge. When there is an edge between two task modules in G_T, the two modules are said to be *related* to each other. Similarly, a multicomputer system can be represented by an undirected graph called the *processor graph*, $G_P = (V_P, E_P)$, where V_P is the set of processors in the system and $E_P \subseteq V_P \times V_P$ is the set of edges representing communication links between processors.

To reduce the inter-processor communication, a set of task modules are usually required to be assigned to a set of processors in such a way that the communication delay between any two related task modules must be kept low. One way to accomplish this is to limit the dilation to a small number so that two related task modules may be assigned to those processors located physically close to each other. Recall that the dilation means the maximal number of hops between two processors to which two related task modules are assigned. An assignment is said to be *acceptable* if its dilation is less than or equal to a prespecified integer value.

The problem of deriving an "optimal" (in the sense of, for example, load balancing or minimization of job execution time) task assignment is very hard and usually requires heuristic

solutions [69, 31]. The task assignment problem is formulated in [77] as a state-space search problem and solved by the A^* algorithm [65]. However, without the knowledge of the number of acceptable assignments for given G_T and G_P, one cannot tell the size of the state-space to be searched in the A^* algorithm. Note that such an algorithm often requires a large number of evaluations of a complex heuristic function.

As will be pointed out later, the knowledge of the number of acceptable assignments can be used not only for providing a simplified state-space search but also for reducing the size of the state-space to be searched. Although a search method using this knowledge may reach a suboptimal goal node instead of the optimal one, it requires much less computation cost and, thus, provides a useful insight into the state-space search. In addition, the knowledge of the number of acceptable assignments and its relation with the processor and task graphs can play an important role in the design of a distributed computing system. In other words, one can derive the system's structure from this knowledge by maximizing the number of acceptable assignments for a given set of cooperating task modules. For the reasons mentioned above, we shall concentrate in this chapter on obtaining the number of acceptable task assignments for given G_T and G_P.

Notice that the necessity of fast communication between two related task modules has usually made it important to assign them to either a single processor or two adjacent processors, i.e., the dilation of an assignment is kept less than or equal to one. Let $N(G_T, G_P)$ denote the number of acceptable assignments under this constraint. As it will be pointed out later, our results on $N(G_T, G_P)$ can be extended and applied to the completely general case, i.e., those assignments with the dilation greater than one. Thus, without loss of generality, we shall henceforth address only the formulation and application of $N(G_T, G_P)$. Unless mentioned otherwise, in what follows, an acceptable assignment is referred to as an assignment

with the dilation less than or equal to one.

To facilitate our discussion, the task assignment problem can be transformed and stated formally as follows. Given the task graph G_T and processor graph G_P, we want to label nodes in G_T with the nodes in G_P in such a way that each node in G_T is labeled with exactly one node from G_P and every pair of adjacent nodes in G_T is labeled with either a single node or two adjacent nodes in G_P. This constraint will be termed *adjacency requirement* and every labeling satisfying the adjacency requirement is called an *acceptable labeling*. Actual task assignment is to choose from the set of acceptable labelings an optimal one that minimizes/maximizes the associated criterion function. Note that we are addressing the problem of determining the number of acceptable assignments, rather than the determination of task assignments themselves. This fact distinguishes our work from other related works [19,77,8,23,68].

This chapter is organized as follows. Section 5.1 deals with the derivation of $N(G_T, G_P)$. First, we derive the bounds of $N(G_T, G_P)$, when G_P and G_T are arbitrary. Then, some important special cases are treated: (i) when G_T is a tree and G_P is arbitrary, and (ii) when G_T is a tree and G_P is an n-regular graph. For case (i), we shall develop a recursive formula to obtain the exact number of $N(G_T, G_P)$. (ii) is a special case of (i) for which the exact number of acceptable assignments can be determined in a closed-form. Also, we shall determine expressions for the case when G_P or G_T is restricted to a certain family of graphs. Three application examples are presented in Section 5.2 to illustrate how the knowledge of the number of acceptable assignments can be used. In light of these examples, some remarks on the task assignment problem are also made. More importantly, it is shown that our results are extensible to the completely general case, i.e., those assignments with dilations greater than one. The results presented in this chapter have also been reported in [80].

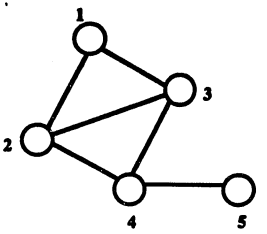
5.1. The Number of Acceptable Assignments

The necessary notation and definitions are introduced in Section 5.1.1. The bounds of $N(G_T, G_P)$, when G_P and G_T are arbitrary are derived in Section 5.1.2. Section 5.1.3 deals with the case when G_T is a tree and G_P is arbitrary. Some restricted cases are given in Section 5.1.4.

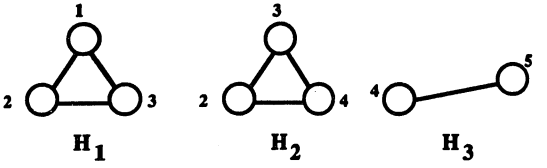
5.1.1. Preliminary

A graph G_α is said to be a *spanning subgraph* of another graph G_β if $V_\alpha = V_\beta$ and $E_\alpha \subseteq E_\beta$ [38]. A *complete subgraph* of a graph G is a subgraph which is a complete graph, and a *clique* is a maximal complete subgraph which is not a proper subgraph of any other complete subgraph of G. Thus, any graph can be viewed as the union of all of its cliques. Let H_i, $1 \le i \le r$, denote the cliques in a graph G, where r is the number of cliques in G. The *redundance* sets of G are defined as $B_i = H_i \cap \{\bigcup_{j=1}^{i-1} H_j\}$ for $1 \le i \le r$. Notice that the determination of redundance sets depends on the ordering of cliques. For example, the graph in Figure 5.1a is the union of its cliques in Figure 5.1b, where $H_1=\{1, 2, 3\}$, $H_2=\{2, 3, 4\}$ and $H_3=\{4, 5\}$. The redundance sets of this graph are: $B_1=\emptyset$, $B_2=\{2, 3\}$ and $B_3=\{4\}$.

The symbol U_m is used to denote an m-dimensional vector of which all entries are one, and P_n, C_n, S_n and K_n to denote respectively a path, cycle, star and complete graph with n nodes [38]. Examples for P_4, C_4, S_4, K_4 and Q_3 are given in Figure 5.2. Recall that Q_n is used to denote an n-dimensional cube. R_n denotes an n-regular graph in which every node has the same degree n. Unless explicitly specified otherwise, every vector referred to in this chapter is treated as a column vector of positive integers, and all graphs are assumed to be connected. In addition, the following definitions are necessary to proceed with our discussion.



(a)



(b)

Figure 5.1. Decomposition of a graph into its components.

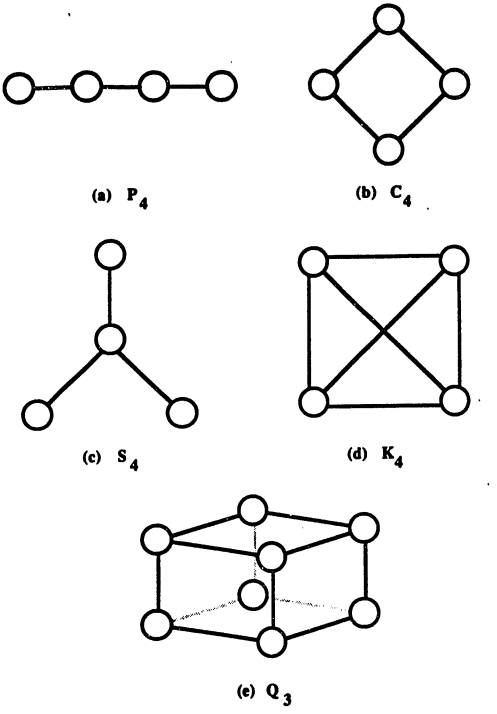


Figure 5.2. Various example graphs.

Definition 5.1: The *adjacency matrix*, $\mathbf{M} = [M_{ij}]$, of a labeled graph G with m nodes is an $m \times m$ matrix in which $M_{ij} = 1$ if node i is adjacent to node j in G or i = j, and $M_{ij} = 0$ otherwise.

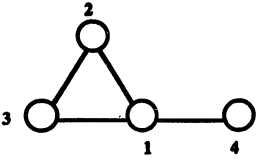
For convenience, in the rest of the chapter we shall use an $m \times m$ non-negative symmetric matrix $\mathbf{A} = [A_{ij}]$ to denote the adjacency matrix of a processor graph G_P, where $m = |V_P|$.

Definition 5.2: Among all acceptable labelings of G_T with the nodes of G_P, let $D_i(k)$ represent the number of acceptable labelings in which the node $n_i \in V_T$ is labeled with the value k, i.e., assigned to processor k in V_P. Then, for each node $n_i \in V_T$, the vector $\mathbf{D}_i = [D_i(1), D_i(2), \cdots, D_i(m)]^T$ is called the *distribution vector* of n_i, where T denotes 'transpose.'

Definition 5.3: The *attaching function* associated with the adjacency matrix A, $\mathbf{f}_A$: $\mathbf{I}^m \rightarrow \mathbf{I}^m$, is defined as $\mathbf{f}_A(\mathbf{V}) = \mathbf{AV}$, $\forall\ \mathbf{V} \in \mathbf{I}^m$ where $\mathbf{I}^m$ is the set of all m-dimensional vectors of positive integers.

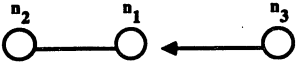
It can be verified that if $\mathbf{D}_i$ is the distribution vector of a task node n_i in G_T and if G_T^* is the resulting graph by attaching a new node y to n_i, then $\mathbf{f}_A(\mathbf{D}_i)$ is the distribution vector of y in G_T^*. For the example processor graph in Figure 5.3a we get $\mathbf{D}_1 = [4, 3, 3, 2]^T$, which means there are 4 ways to label n_1 with processor 1, 3 ways to label n_1 with processor 2, and so on. After n_3 is attached to n_1, we have $\mathbf{D}_3 = \mathbf{f}_A([4, 3, 3, 2]^T) = [12, 10, 10, 6]^T$ in Figure 5.3c. Notice that, to satisfy the adjacency requirement, n_3 in G_T^* can be labeled with processor 2 only when n_1 in G_T was labeled with any of processor 1, processor 2 and processor 3. Then, we have $D_3(2) = 4 + 3 + 3 = 10$. The other entries in $\mathbf{D}_3$ can be obtained similarly.

Also, introduced are the following four definitions, which will be very useful in determining $N(G_T, G_P)$ when G_T is a tree.



(a) G_P

(b) G_T



(c) G_T^*

Figure 5.3. An illustrative example for the attaching function.

Definition 5.4: The *product* of two vectors, denoted by $\odot$, is defined as $V_3 = V_1 \odot V_2$ iff $V_3(k) = V_1(k)V_2(k)$ for $1 \le k \le m$, $\forall\ V_1, V_2, V_3 \in I^m$, where $V_i(k)$ denotes the k-th element of the vector V_i for i = 1, 2, 3.

Clearly, the product of vectors is associative and commutative. We shall use $\prod_{i=1}^{q} V_i$ to denote the product of q vectors. Note that this operation is not the same as the conventional inner product of vectors, since it results in a vector, rather than a scalar.

Definition 5.5: The *multiplication* of two vectors associated with the adjacency matrix A, denoted by $*_A$, is defined as:

$$V_1 *_A V_2 = V_1 \odot f_A(V_2) \quad \forall\ V_1, V_2 \in I^m.$$

Definition 5.6: The *sorted vector* of a vector $V \in I^m$, denoted by $V^* \in I^m$, is a vector whose components, $V^*(i)$, $1 \le i \le m$, are a descending permutation of the components of V.

Definition 5.7: The *weight of a vector*, $W: I^m \to I$, is defined as:

$$W(V) = \sum_{i=1}^{m} V(i), \quad \forall\ V \in I^m.$$

5.1.2. The case of arbitrary processor and task graphs

The following lemma immediately follows from the adjacency requirement in task assignment.

Lemma 5.1: (i) $N(G_T, G_P) \le N(G_T, G_P')$ if G_P is a spanning subgraph of G_P'.

(ii) $N(G_T, G_P) \ge N(G_T', G_P)$ if G_T is a spanning subgraph of G_T'.

By (ii) of this lemma, the inequality $N(G_T, G_P) \le N(S(G_T), G_P)$ always holds, where $S(G_T)$ is an arbitrary spanning tree of G_T. Moreover, we have the following theorem.

Theorem 5.1: For any arbitrary G_P and G_T, there exist the following bounds of $N(G_T, G_P)$, which are independent of $|E_T|$:

(i) $N(G_T, G_P) \geq N(K_{|V_T|}, G_P) = \sum_{i=1}^{r} (|H_i|^{|V_T|} - |B_i|^{|V_T|})$

(ii) $N(G_T, G_P) \leq N(S(G_T), G_P) \leq N(S_{|V_T|}, G_P) = \sum_{i=1}^{|V_P|} (d_i + 1)^{|V_T|} - 1$,

where, as before, H_i and B_i, $1 \leq i \leq r$, are respectively the clique and redundance sets of G_P, d_i is the degree of node i in G_P and $S_{|V_T|}$ is a star with $|V_T|$ nodes.

Proof: (i) Since the complete graph $K_{|V_T|}$ possesses the maximal number of edges among all the graphs with $|V_T|$ nodes, the inequality $N(G_T, G_P) \geq N(K_{|V_T|}, G_P)$ follows from (ii) of Lemma 5.1. Every node in $K_{|V_T|}$ is adjacent to all the other nodes; to satisfy the adjacency requirement, all the nodes in $K_{|V_T|}$ must be labeled with nodes within one clique in G_P. There are $|H_i|^{|V_T|}$ ways to label G_T with nodes in the clique set H_i of G_P, and, thus, the total number of such labelings can be obtained by adding up all $|H_i|^{|V_T|}$, $1 \leq i \leq r$, where r is the number of cliques in G_P. However, the number of labelings with a clique set H_i contains double counts for those labelings in which all the nodes in $K_{|V_T|}$ are labeled with nodes in the redundant set B_i. The redundant counts must be removed by subtracting $|B_i|^{|V_T|}$ from $|H_i|^{|V_T|}$, and, thus, (i) follows.

Because the proof of (ii) requires more bases to cover, we shall complete the proof of (ii) after Theorem 5.2. **Q.E.D.**

Notice that when G_P is a complete graph, the number of acceptable assignments is $|V_P|^{|V_T|}$ regardless of the type of G_T. This fact implies that the tightness of the bounds depends strongly on the structure of G_P. From Theorem 5.1, we have the following corollary

for the lower bound of $N(G_T, G_P)$ when G_P is restricted to a hypercube or a cycle. Recall that Q_n denotes an n-dimensional cube and C_m is a cycle with m nodes.

Corollary 5.1.1:

(i). $N(K_h, Q_n) = 2^{n+h-1}n - 2^n(n-1)$,

(ii). $N(K_h, C_m) = 2^h m - m$.

Proof: Since there is no K_3 subgraph in Q_n, all the nodes in K_h must be labeled with either a single node or two adjacent nodes in Q_n. There are 2^h ways to label K_h with each pair of adjacent nodes in Q_n, and the number of edges in Q_n is $2^{n-1}n$. Therefore, when each edge in Q_n is considered separately, the total number of acceptable labelings is $2^{n-1}n2^h$. However, the labeling in which all the nodes in K_h are labeled with the same node from Q_n occurs n times. Thus, we have to remove the redundant counts by subtracting $2^n(n-1)$ from $2^{n-1}n2^h$, leading to $N(K_h, Q_n) = 2^{n+h-1}n - 2^n(n-1)$.

It can be easily seen that (ii) is valid when m=3. Note that there is no K_3 subgraph in C_m, $\forall$ m>3. Thus, (ii) can be proved similarly. **Q.E.D.**

As shown above, one can derive only bounds[4] of $N(G_T, G_P)$ when G_T and G_P are both arbitrary graphs. However, when G_T is restricted to a tree, $N(G_T, G_P)$ can be expressed in a recursive-form as will be shown in the following subsection. Moreover, it will be shown in Section 5.1.4 that $N(G_T, G_P)$ can be expressed in a closed-form when G_T is a tree and G_P is an n-regular graph, R_n.

[4]These bounds are necessarily loose because of the wide range of structural variations in the processor and task graphs.

5.1.3. The case when G_T is a tree

To derive the recursive formula for $N(G_T, G_P)$ when G_T is a tree and G_P is arbitrary, we must convert G_T to a rooted tree by choosing an arbitrary node of G_T as the root. Let us define the *carrying vector* of a node of G_T as follows.

Definition 5.8: The *carrying vector* of a node $n_i \in V_T$, denoted by $\mathbf{Y}_i$, is defined in a recursive-form,

$$\mathbf{Y}_i = \begin{cases} \mathbf{U}_m & \text{if } n_i \text{ is a leaf,} \\ \prod_{n_j \in C(n_i)} \mathbf{f}_A(\mathbf{Y}_j) & \text{otherwise,} \end{cases} \tag{5.1}$$

where $C(n_i)$ represents the set of children of the node n_i.

As we shall prove later in Theorem 5.2, the carrying vector is so defined that we can determine the distribution vector of any node, say n_k, just by rearranging the tree to make n_k the root and then computing the carrying vector of the node n_k by Eq. (5.1).

Let n_1 and n_2 be two nodes with distribution vectors $\mathbf{D}_1$ and $\mathbf{D}_2$ in task graphs G_1 and G_2, respectively. Suppose G' is the resulting graph by adding a new edge between n_1 and n_2 to connect G_1 and G_2. Then, the distribution vectors of the two nodes n_1 and n_2 in G' can be determined by the following lemma.

Lemma 5.2: Let $\mathbf{D}_1'$ and $\mathbf{D}_2'$ denote respectively the resulting distribution vectors of task nodes n_1 and n_2 after connecting n_1 and n_2 with a new edge. Then,

(i) $\mathbf{D}_1' = \mathbf{D}_1 *_A \mathbf{D}_2$

(ii) $\mathbf{D}_2' = \mathbf{D}_2 *_A \mathbf{D}_1$

where $\mathbf{A}$ is the adjacency matrix of G_P.

Proof: Consider part (i). When n_1 in G_1 and n_2 in G_2 are labeled respectively with nodes i and j in G_P, $A_{ij} = A_{ji} = 1$ is the necessary and sufficient condition that this labeling is still acceptable for the resulting graph G'. The number of labelings of G_2 which are still acceptable after connecting n_1 and n_2 is $\sum_{j=1}^{m} A_{ji} D_2(j)$. Since the number of different acceptable labelings of G_1 in which n_1 is labeled with i is $D_1(i)$, we get $D_1'(i) = D_1(i) \sum_{j=1}^{m} A_{ji} D_2(j)$ and part (i) follows. Part (ii) can be proved similarly. **Q.E.D.**

Thus, we have the following important result.

Theorem 5.2: The carrying vector of the root node of a tree is the same as its distribution vector, i.e., $\mathbf{D}_r = \mathbf{Y}_r$ if n_r is the root of a tree.

Proof: We prove this theorem by induction. Obviously, the theorem holds for a trivial tree (i.e., a tree with only one node). In that case, both the distribution and carrying vectors are $\mathbf{U}_m$, where $m = |V_r|$ as before.

Assume that the theorem holds for all the children of a node, say n_r. Let n_{r_i}, $1 \le i \le c$, denote the node n_r's children, i.e., $C(n_r) = \{ n_{r_i} \mid 1 \le i \le c \}$, where $c = |C(n_r)|$. Then, from Lemma 5.2 the distribution vector of n_r with only one child n_{r_1} is $\mathbf{U}_m *_A \mathbf{Y}_{r_1}$. Thus, by attaching one more child at a time, the distribution vector of n_r with all its children attached becomes:

$$\begin{aligned}
\mathbf{D}_r &= \mathbf{U}_m *_A \mathbf{Y}_{r_1} *_A \mathbf{Y}_{r_2} *_A \cdots *_A \mathbf{Y}_{r_c} \\
&= [\mathbf{U}_m \odot (A\mathbf{Y}_{r_1})] \odot (A\mathbf{Y}_{r_2}) \odot \cdots \odot (A\mathbf{Y}_{r_c}) \\
&= \prod_{i=1}^{c} A\mathbf{Y}_{r_i}
\end{aligned}$$

$$= \prod_{n_j \in C(n_r)} f_A(Y_j)$$

$= Y_r$. **Q.E.D.**

Given an arbitrary G_P, one can compute the carrying vector of all nodes in a rooted tree G_T by applying Eq. (5.1) recursively, and determine the number of acceptable labelings from the corollary below.

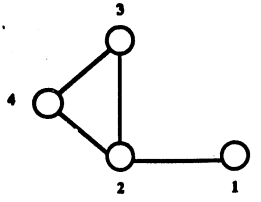
Corollary 5.2.1: $\forall\ n_i \in V_T$, $W(Y_i) = N(T(n_i), G_P)$, where $T(n_i)$ is the tree formed by the node n_i and its descendants.

Proof: Suppose $T(n_i)$ is the task tree, then we have $D_i = Y_i$ from Theorem 5.2. This corollary follows from the fact that $W(D_i) = N(T(n_i), G_P)$. **Q.E.D.**

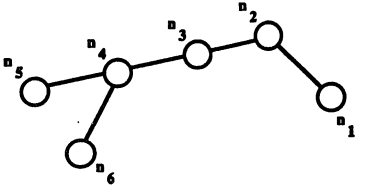
The computational complexity in obtaining $N(G_T, G_P)$, when G_T is a tree, can be determined as follows. By Lemma 5.2 and Theorem 5.2, $N(G_T, G_P)$ can be obtained by calculating the carrying vector of each node in the tree with Eq. (5.1). The complexity in obtaining $f_A(Y_j)$ from given A and Y_j is $O(m^2)$, where $m = |V_P|$, $Y_j \in I^m$ and A is an $m \times m$ matrix. Note that the complexity of the operation $\odot$ is $O(m)$. Thus, the complexity in calculating $N(G_T, G_P)$, when G_T is a tree, is $O(|V_T|)O(m^2 + m) = O(|V_T|m^2)$.

To illustrate the ideas presented thus far, consider the example processor and task graphs shown in Figures 5.4a and 5.4b. We have the following adjacency matrix for G_P in Figure 5.4a.

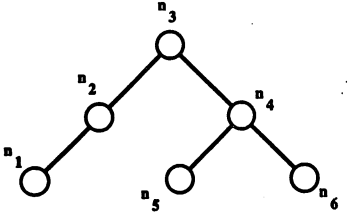
$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}.$$



(a) Processor graph, G_P



(b) Task graph, G_T



(c) A rooted tree derived from G_T

Figure 5.4. Example processor and task graphs.

Suppose n_3 in G_T is chosen as the root of the tree. Then, we get the rooted tree in Figure 5.4c where n_1, n_5 and n_6 are leaf nodes (i.e., $Y_1 = Y_5 = Y_6 = [1\ 1\ 1\ 1]^T$), and $C(n_3) = \{n_2, n_4\}$, $C(n_2) = \{n_1\}$ and $C(n_4) = \{n_5, n_6\}$. Thus, we get

$$Y_2 = f_A(Y_1) = AY_1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 3 \\ 3 \end{bmatrix}$$

$$Y_4 = (AY_5) \odot (AY_6) = \begin{bmatrix} 2 \\ 4 \\ 3 \\ 3 \end{bmatrix} \odot \begin{bmatrix} 2 \\ 4 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 4 \\ 16 \\ 9 \\ 9 \end{bmatrix}$$

$$Y_3 = (AY_2) \odot (AY_4) = \begin{bmatrix} 6 \\ 12 \\ 10 \\ 10 \end{bmatrix} \odot \begin{bmatrix} 20 \\ 38 \\ 34 \\ 34 \end{bmatrix} = \begin{bmatrix} 120 \\ 456 \\ 340 \\ 340 \end{bmatrix}$$

$W(Y_3) = 120 + 456 + 340 + 340 = 1256.$

From Figure 5.4a, we get $H_1=\{2, 3, 4\}$, $H_2=\{1, 2\}$, and then $B_1=\emptyset$ and $B_2=\{2\}$. It can be verified that $3^6 + 2^6 - 1^6 = 792 < 1256 < 2^5 + 4^5 + 3^5 + 3^5 = 1542$, which agrees with the bounds in Theorem 5.1.

In what follows, we shall prove part (ii) of Theorem 5.1. To facilitate the proof, we need the following definition [56]. Also, recall that V^* denotes the sorted vector of V.

Definition 5.9: Let V_1 and V_2 be two m-dimensional vectors. V_1 is said to be *weakly submajorized* by V_2, denoted by $V_1 <_w V_2$, if $\sum_{i=1}^{k} V_1^*(i) \le \sum_{i=1}^{k} V_2^*(i)$ for $1 \le k \le m$.

Then, we have the following three propositions.

Proposition 5.1: If $V_1 <_w V_2$, then $V_1^* \odot V_3^* <_w V_2^* \odot V_3^*$ $\forall\ V_1, V_2, V_3 \in I^m$.

Proof: Let $V_1^*=[a_1, a_2, \cdots, a_m]^T$, $V_2^*=[b_1, b_2, \cdots, b_m]^T$, $V_3^*=[c_1, c_2, \cdots, c_m]^T$ and $\delta_i = V_1^*(i) - V_2^*(i) = a_i - b_i$. Since $V_1 <_w V_2$, $\sum_{i=1}^{k}\delta_i \le 0$ for $1 \le k \le m$. We obtain

$$\sum_{i=1}^{k}c_ia_i - \sum_{i=1}^{k}c_ib_i = \sum_{i=1}^{k}c_i(a_i - b_i)$$

$$= c_1\delta_1 + c_2\delta_2 + \sum_{i=3}^{k}c_i\delta_i$$

$$\le c_2(\delta_1 + \delta_2) + \sum_{i=3}^{k}c_i\delta_i \quad \text{(Since } \delta_1 \le 0 \text{ and } \{c_i\} \text{ is decreasing.)}$$

$$\le c_u\sum_{i=1}^{u}\delta_i + \sum_{i=u+1}^{k}c_i\delta_i$$

$$\le c_k\sum_{i=1}^{k}\delta_i \le 0,$$

and this proposition follows. **Q.E.D.**

Proposition 5.2: Given a vector $V \in I^m$ and a decreasing sequence of node degrees, $\{d_i\}_{i=1}^m$, in a graph with an adjacency matrix A, if $V <_w [(d_1+1)^n,(d_2+1)^n, \cdots,(d_m+1)^n]^T$, then $f_A(V) <_w [(d_1+1)^{n+1},(d_2+1)^{n+1}, \cdots, (d_m+1)^{n+1}]^T$.

Proof: Let $Z = f_A(V)$. Note that $W(Z) = W(Z^*)$ is the summation of all the elements in the set $L = \{V(1), \cdots, V(1), V(2), \cdots, V(2), \cdots, V(m), \cdots, V(m)\}$ in which V(i) appears d_i+1 times. Since $\sum_{i=1}^{k}Z^*(i)$ is the summation of no more than $\sum_{i=1}^{k}(d_i+1)$ elements from L, we obtain $\sum_{i=1}^{k}Z^*(i) \le \sum_{i=1}^{k}V^*(i)(d_i+1) \le \sum_{i=1}^{k}(d_i+1)^{n+1}$ (by Proposition 5.1). Thus, the proposition follows. **Q.E.D.**

Proposition 5.3: Let $\{p_i\}_{i=1}^m$ be a decreasing sequence of positive integers, and V_1, V_2 and V_3 be m-dimensional vectors such that for some positive integers r and s, $V_1 <_w [p_i^r,$

$p_2^r, \cdots, p_m^r]^T$ and $V_2 <_w [p_1^s, p_2^s, \cdots, p_m^s]^T$. Then, $V_3 = V_1 \odot V_2 <_w [p_1^{r+s}, p_2^{r+s}, \cdots, p_m^{r+s}]^T$.

Proof: For $1 \le k \le m$, $\sum_{i=1}^{k} V_3^*(i) \le \sum_{i=1}^{k} V_1^*(i) V_2^*(i)$

$$\le \sum_{i=1}^{k} V_1^*(i) p_i^s \quad \text{(by Proposition 5.1)}$$

$$\le \sum_{i=1}^{k} p_i^{r+s} \quad \text{(by Proposition 5.1).} \qquad \textbf{Q.E.D.}$$

Without loss of generality, let $\{d_i\}_{i=1}^{m}$ be a decreasing sequence of node degrees in G_P. Then, part (ii) of Theorem 5.1 can be proved as follows.

Proof for part (ii) of Theorem 5.1: The first inequality of (ii) in Theorem 5.1 follows directly from Lemma 5.1. To obtain the equality, $N(S_{|V_T|}, G_P) = \sum_{i=1}^{|V_P|} (d_i + 1)^{|V_T|-1}$, consider the case where one more node is to be attached to the central node of a star at a time. Then, this equality follows immediately. Next, we want to prove the second inequality.

We claim that the carrying vector of the root node of a tree with n nodes is weakly sub-majorized by $[(d_1+1)^{n-1}, (d_2+1)^{n-1}, \cdots, (d_m+1)^{n-1}]^T$ and, then, the required result follows from Theorem 5.2. We prove this claim by induction. Clearly, for a trivial tree, $\sum_{i=1}^{k} U_m(i) \le \sum_{i=1}^{k} 1$, $1 \le k \le m$. Next, let s_j be the number of nodes in the tree $T(n_j)$ which is formed by n_j and its descendants. Assume that $Y_j <_w [(d_1+1)^{s_j-1}, (d_2+1)^{s_j-1}, \cdots, (d_m+1)^{s_j-1}]^T$. Then, by Proposition 5.2 we have $f_A(Y_j) <_w [(d_1+1)^{s_j}, (d_2+1)^{s_j}, \cdots, (d_m+1)^{s_j}]^T$. Since $Y_i = \prod_{n_j \in C(n_i)} f_A(Y_j)$, by Proposition 5.3 we obtain $Y_i <_w [(d_1+1)^{s_i-1}, (d_2+1)^{s_i-1}, \cdots, (d_m+1)^{s_i-1}]^T$, where $C(n_i)$ is the set of children of the node n_i and $s_i = \sum_{n_j \in C(n_i)} s_j + 1$. The claim is thus proved by induction and the inequality $N(G_T, G_P) \le \sum_{i=1}^{|V_P|} (d_i + 1)^{|V_T|-1}$ follows.

Q.E.D.

5.1.4. Some restricted cases

In a more restricted case when G_T is a tree and G_P is an n-regular graph, $N(G_T, G_P)$ can be expressed in a closed-form as given in the following corollary.

Corollary 5.2.2: If G_P is an n-regular graph and G_T is an arbitrary tree, then

$$N(G_T, R_n) = |V_P|(n+1)^{|V_T|-1}.$$

Proof: If **A** is the adjacency matrix of an n-regular graph, we have $W(\mathbf{f}_A(V)) = (n+1)W(V)$, $\forall\ V \in I^m$. If n_s is an isolated node, then $W(\mathbf{D}_s) = W(U_m) = |V_P|$. We can construct any tree by starting with a single node and attaching one node at a time, thereby proving this corollary. **Q.E.D.**

Due to the diversity of network structures, the problem of determining $N(G_T, G_P)$ however becomes very complicated when G_T is neither a tree nor a complete graph. Although one can derive recursive or closed form expressions for $N(G_T, G_P)$ when G_T and G_P are restricted to some family of graphs, it is still extremely difficult to determine the general formula of $N(G_T, G_P)$. The procedure of determining $N(C_h, Q_n)$ is presented below to demonstrate the associated difficulty.

To determine the expression of $N(C_h, Q_n)$, consider an alternative approach to the determination of $N(P_h, Q_n)$ without applying Corollary 5.2.2. Suppose the two end nodes of a (task) path P_h are assigned to processors p_1 and p_2 in Q_n, the distance between which is k. Consider the case where one more task node is to be attached to the end node labeled with p_2. Clearly, the attaching task node can be assigned to n+1 possible processor nodes in Q_n. From

the adjacency requirement and the structure of Q_n, we know that k of these processors have a distance k–1 from p_1, n–k of them have a distance k+1 from p_1 and only one of them has a distance k from p_1. That is, the relationship of the two end nodes of P_{h+1} can be determined from the relationship of the two end nodes of P_h. More formally, let us define the sequence of vectors $\{a_{i,0}, a_{i,1}, \cdots, a_{i,n}\}$ such that

$$a_{1,0} = 2^n, \quad a_{1,i} = 0, \quad 1 \le i \le n,$$

$$a_{k,j} = (n-j+1)a_{k-1,j-1} + a_{k-1,j} + (j+1)a_{k-1,j+1}, \quad k \ge 2, \quad 1 \le j \le n-1,$$

$$a_{k,0} = a_{k-1,0} + a_{k-1,1}, \qquad a_{k,n} = a_{k-1,n-1} + a_{k-1,n}, \qquad k \ge 2.$$

Note that this sequence of vectors is so defined that $a_{i,j}$ is the number of acceptable labelings of P_i with Q_n in which the distance between the two processors assigned to the two end nodes in P_i is j. Using this sequence, we have the following lemma which determines $N(C_h, Q_n)$.

Lemma 5.3: (a) $N(P_h, Q_n) = \sum_{i=0}^{n} a_{h,i} = 2^n(n+1)^{h-1}$.

(b) $N(C_h, Q_n) = a_{h,0} + a_{h,1}$.

Proof: The acceptable labelings of P_h with Q_n, in which the distance between the two processors assigned to the two end nodes in P_h is j, come from the following three cases.

Case 1: Adding one more node to P_{h-1} in which the distance between the two processors assigned to the two end nodes is j–1.

Case 2: Adding one more node to P_{h-1} in which the distance between the two processors assigned to two end nodes is j.

Case 3: Adding one more node to P_{h-1} in which the distance between the two processors assigned to two end nodes is j+1.

For these three cases there are n–j+1, 1 and j+1 possible P_{h-1}'s, respectively. Thus, part (a) follows from the definition of the sequence of vectors, $\{a_{i,0}, a_{i,1}, \cdots, a_{i,n}\}$.

Clearly, C_h can be obtained by adding one more edge between the two end nodes of a P_h. Thus, part (b) follows. **Q.E.D.**

Figure 5.5 shows how to determine $N(P_i, Q_n)$ and, thus, $N(C_i, Q_n)$ with the above method when n=2 and n=3, respectively. The entry in row P_i and column Δ_j, denoted by $a_{i,j}$, means the number of acceptable labelings of P_i with Q_n in which the distance between the processors assigned to two end nodes of P_i is j. It can be easily verified that the numbers $N(P_i, Q_n)$, $i \geq 1$, agree with the result of Corollary 5.2.2. It is interesting to analyze the complexity of calculating $N(C_h, Q_n)$. From Figure 5.5 and the recursive definition of $\{a_{i,0}, a_{i,1}, \cdots, a_{i,n}\}$, it requires 2 multiplications and 2 additions to determine each entry in row P_i of Figure 5.5 from entries in row P_{i-1}, and there are n+1 entries in each row, meaning that the complexity of calculating all entries in a row is $O(n)$. Therefore, the complexity of obtaining $N(C_h, Q_n)$ is $O(nh)$.

Notice that even in the restricted case of G_P is a Q_n and G_T is a C_h, we have to appeal to some nontrivial recursive formula. Naturally, the difficulty in determining $N(G_T, G_P)$ increases with the irregularity of the graphs involved. The main results in this section are summarized in Table 5.1.

5.2. Application, Remarks and Extension

In this section, three application examples are presented to demonstrate the utility of the knowledge of $N(G_T, G_P)$. In light of these examples, some remarks are also made to indicate the complexity of the task assignment problem. More importantly, our results on $N(G_T, G_P)$

When n = 2	Δ_0	Δ_1	Δ_2	$N(P_i, Q_2)$
P_1	4			4
P_2	4	8		12
P_3	12	16	8	36
P_4	28	56	24	108
P_5	84	160	80	324

When n = 3	Δ_0	Δ_1	Δ_2	Δ_3	$N(P_i, Q_3)$
P_1	8				8
P_2	8	24			32
P_3	32	48	48		128
P_4	80	240	144	48	512
P_5	320	768	768	192	2048

Figure 5.5. Examples of $N(P_i, Q_n)$ for n=2 and n=3.

G_T \ G_P	arbitrary	R_n	Q_n
K_h	$\sum_{i=1}^{r} \lvert H_i \rvert^h - \lvert B_i \rvert^h$ †	$\sum_{i=1}^{r} \lvert H_i \rvert^h - \lvert B_i \rvert^h$	$2^{n+h-1}n - 2^n(n-1)$
S_h	$\sum_{i=1}^{\lvert V_P \rvert} (d_i+1)^{h-1}$ ††	$\lvert V_P \rvert (n+1)^{h-1}$	$2^n (n+1)^{h-1}$
tree	$W(Y_i)$ †††	$\lvert V_P \rvert (n+1)^{\lvert V_T \rvert - 1}$	$2^n (n+1)^{\lvert V_T \rvert - 1}$
arbitrary	lower bound = $\sum_{i=1}^{r} \lvert H_i \rvert^{\lvert V_T \rvert} - \lvert B_i \rvert^{\lvert V_T \rvert}$, upper bound = $\sum_{i=1}^{\lvert V_P \rvert} (d_i+1)^{\lvert V_T \rvert - 1}$		

† H_i , B_i , $1 \le i \le r$, are respectively the component and redundance sets of G_P .

†† $\{d_i\}$ is the degree sequence of G_P.

††† n_i is the arbitrary node in G_T.

Table 5.1. The number of acceptable labelings in various cases.

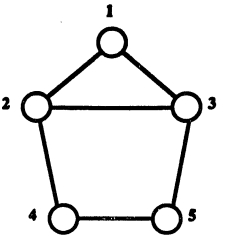
are extended to the completely general case, (i.e., those assignments with dilations greater than one) in which two related task modules in G_T can be assigned to *any* two processors in G_P (which are not required to be adjacent to each other).

5.2.1. Application Examples

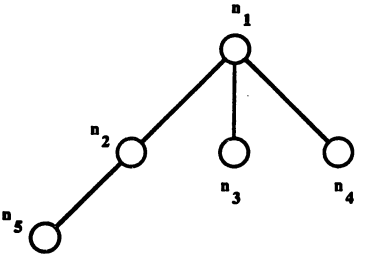
Example 5.1: Consider the processor graph G_P and task graph G_T shown in Figures 5.6a and 5.6b, respectively. Let $G_P\backslash(i,j)$ denote the graph resulting from the removal of the edge (i,j) from G_P. Then, using the results in Section 5.1.3, we can calculate $N(G_T, G_P) = 727$, $N(G_T, G_P\backslash(1,3)) = 477$, $N(G_T, G_P\backslash(4,5)) = 563$, $N(G_T, G_P\backslash(2,3)) = 405$, and $N(G_T, G_P\backslash(3,5)) = 489$. Therefore, as far as the number of acceptable assignments is concerned, the edge (2,3) is the most critical since its removal will cause the largest decrease in the number of acceptable labelings.

Consider the case when a new edge or communication link is to be added in G_P. Let $G_P+(i,j)$ denote the graph resulting from the addition of an edge between nodes i and j in G_P. Then, we obtain $N(G_T, G_P+(1,4)) = 1091$ and $N(G_T, G_P+(3,4)) = 1203$. This implies a higher increase in the number of acceptable labelings by adding edge (3,4) than by adding edge (1,4). Thus, using $N(G_T, G_P)$ one can determine the edge to be added for a maximal increase in the number of acceptable labelings.

Example 5.2: Consider the example processor graph G_P and task graph G_T shown in Figures 5.7a and 5.7b, respectively. Again applying the procedure in Section 5.1.3, we can construct an *enumeration tree* as shown in Figure 5.8, where the number in the square associated with each node represents the number of all acceptable labelings subject to the partial labeling made already for the node and its predecessors.

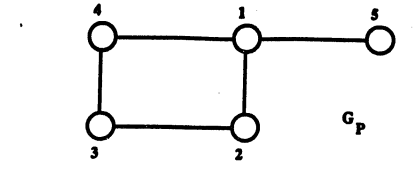


(a) Processor graph, G_P

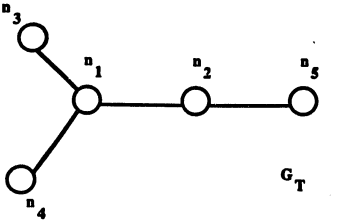


(b) Task graph, G_T

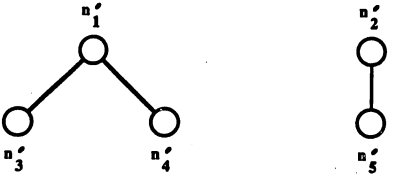
Figure 5.6. Example processor and task graphs.



(a) Processor graph



(b) Task graph



(c) Decomposed task graph

Figure 5.7. An example network.

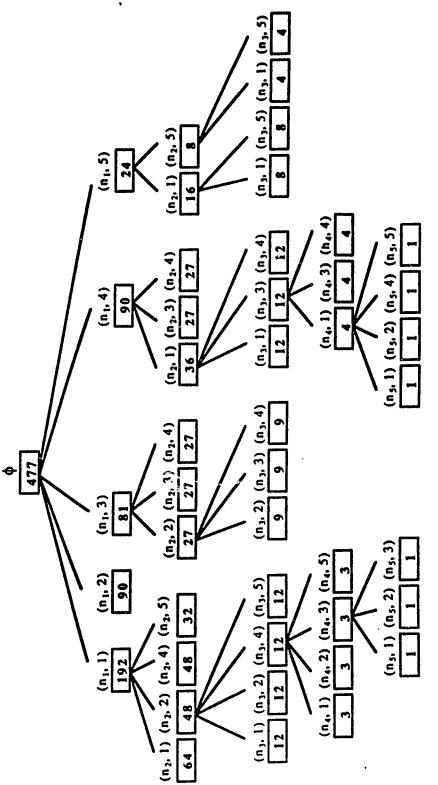


Figure 5.8. Part of the enumeration tree with the associated graph in Figure 5.7.

Notice that the numbers in the squares in the first level (i.e., 192, 90, 81, 90 and 24) are $D_1(i)$, $1 \le i \le 5$, and those in the second or lower levels can be obtained by properly decomposing the task tree and multiplying together the numbers of acceptable labelings in each subtree subject to the partial labeling made already for the corresponding node and its predecessors. For example, to determine the number in the square of the node (n_2, 2) whose immediate predecessor is (n_1, 1), we decompose the task tree into two subtrees as shown in Figure 5.7c and then obtain the number associated with the node (n_2, 2) from $D_1'(1)^* D_2'(2) = 16*3 = 48$.

The enumeration tree can also be used to determine the number of conditional acceptable labelings, $N(G_T, G_P \mid P)$, where $P \subseteq \{(t, p) \mid t \in V_T, p \in V_P\}$. For example, if $P = \{(n_1,5), (n_3, 1)\}$, then $N(G_T, G_P \mid P) = 8 + 4 = 12$. This means that there are 12 acceptable labelings in which task modules n_1 and n_3 are assigned to processors 5 and 1, respectively. Moreover, in the state-space search of the task assignment problem, the enumeration tree provides a good indication for the search status of the current node and can be applied to establish a *guided* search. When the computation cost for the heuristic function is high[5], we can skip some evaluation steps and choose a search route toward an ampler state-space without computing the heuristic function of every offspring. For example, in Figure 5.8 the node (n_1,1) will be chosen since it has the highest potential (i.e., 192 > 90, 81, 24) for containing the optimal solution. This approach is actually based on the fact that a larger number of acceptable labelings implies that unassigned task modules have a better chance to be spread out in the network.

Clearly, when the goal of achieving load balancing is more important than that of reducing the interprocessor communication cost, the likelihood of making a successful guess with the knowledge of $N(G_T, G_P)$ will be increased. This guided search holds practical importance,

[5]For example, the algorithm A* used in [77] requires a large number of evaluations of a complex heuristic function.

since it requires much less search cost and is attractive, especially when we want to reduce the expected search cost and there are many acceptable goal nodes in the state-space.

Example 5.3: In the state-space search, we naturally want to reduce the number of expanded and generated nodes in the worst case [65]. Consider the enumeration tree in Figure 5.8. It is easy to see that the number of nodes in the i-th level is equal to $N(TR_i, G_P)$ where TR_i is the induced subgraph of G_T with the set of nodes $\{n_j \mid n_j \in V_T \text{ and } j \leq i\}$. Note that the total number of nodes in the enumeration tree, $1 + \sum_{i=1}^{|V_T|} N(TR_i, G_P)$, and the number of internal nodes, $1 + \sum_{i=1}^{|V_T|-1} N(TR_i, G_P)$, are respectively the number of generated nodes and the number of expanded nodes in the worst case of the state-space search. For the example task and processor graphs in Figure 5.7, we get $N(TR_1, G_P) = 5$, $N(TR_2, G_P) = 15$, $N(TR_3, G_P) = 47$, $N(TR_4, G_P) = 153$ and $N(TR_5, G_P) = 477$. That is, there are $1 + \sum_{i=1}^{|V_T|-1} N(TR_i, G_P) = 221$ expanded nodes and $1 + \sum_{i=1}^{|V_T|} N(TR_i, G_P) = 698$ generated nodes in the worst case of the state-space search.

Clearly, while $N(TR_{|V_T|}, G_P)$ is just the number of acceptable labelings, the number $\sum_{i=1}^{|V_T|-1} N(TR_i, G_P)$ closely depends on how we encode the task nodes in G_T with n_i, $1 \leq i \leq |V_T|$. In other words, different encodings of the task tree lead to different enumeration trees which usually have different numbers of internal nodes but the same number of leaves. For example, in the case when the task graph in Figure 5.7b is encoded as the one in Figure 5.9, we have $N(TR_1, G_P) = 5$, $N(TR_2, G_P) = 15$, $N(TR_3, G_P) = 47$, $N(TR_4, G_P) = 147$ and $N(TR_5, G_P) = 477$. The maximal numbers of generated and expanded nodes are then reduced to 692 and 215, respectively.

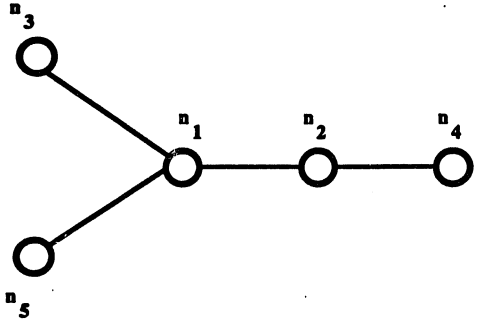


Figure 5.9. A different encoding of the task graph in Figure 5.7b.

It can be verified by enumeration that among all the possible encodings for the task tree in Figure 5.7b, the encoding in Figure 5.9 is the enumeration tree with the minimal number of internal nodes; it minimizes the number of expanded nodes in the worst case of the state-space search when the processor graph is the one in Figure 5.7a. (Such an encoding is termed the *best encoding*.) Improvement in the worst case of the state-space search is not the only advantage of the encoding with a smaller enumeration tree. Since the goal node in the state-space search must be a leaf, searches in the enumeration tree with less internal nodes are naturally expected to have less average number of expanded and generated nodes.

Using the procedure proposed here, one can construct the enumeration tree for each encoding of the task tree and then determine the best encoding off-line to reduce the computation cost of the state-space search.

5.2.2. Remarks

The following remarks are in order to clarify some conjectures which may result from the above examples.

R1. In the first example, the increase of the number of acceptable labelings by adding an edge between two processor nodes with larger degrees may always seem to be greater than that by adding an edge between nodes with smaller degrees. This is not always true. A counter example is shown in Figure 5.10, where G_β and G_γ are obtained by adding edges (1,13) and (7,10) respectively in G_α. Applying the results in Section 5.1.3, we get $N(P_6, G_\beta) = 10134 < N(P_6, G_\gamma) = 10454$. Note that the degrees of nodes 1 and 13 in G_α are 4, whereas those of nodes 7 and 10 in G_α are 3.

R2. We get $N(P_3, G_\beta) = 196 > N(P_3, G_\gamma) = 192$ in Figure 5.10. This means that the edge (1,13) in G_δ is more important than the edge (7,10) when the task graph is P_3, but less

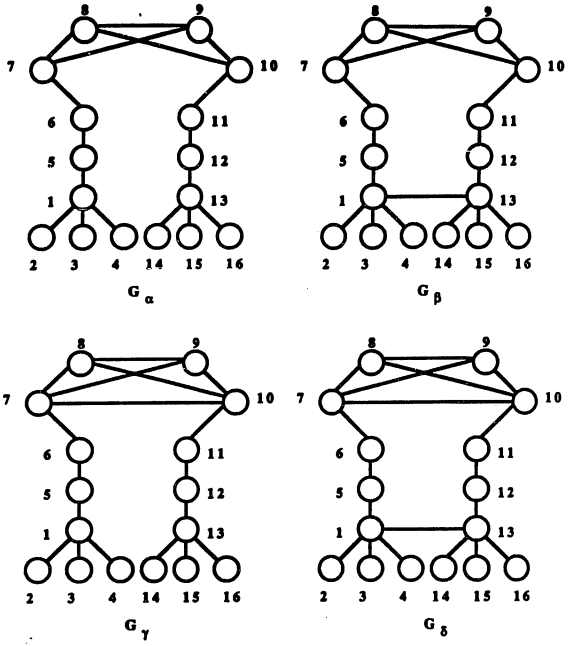


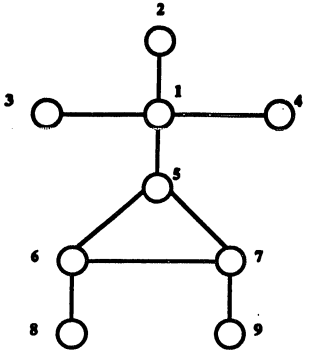
Figure 5.10. A counter example showing that adding an edge between two nodes with larger degrees does not always result in a higher increase in the number of acceptable labelings.

important than the edge (7,10) when the task graph is P_6. This fact not only indicates the importance of each edge in the processor graph depends on the structure of the associated task graph but also shows that $N(G_{T_1}, G_{P_1}) > N(G_{T_1}, G_{P_2})$ does not imply $N(G_{T_2}, G_{P_1}) > N(G_{T_2}, G_{P_2})$ for $G_{T_2} \neq G_{T_1}$.

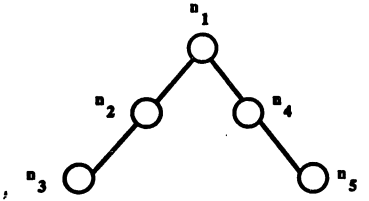
R3. One may also conjecture in Example 5.2 that the processor node which leads to the amplest state-space is always the node with the maximal degree. Again, a counter example is given in Figure 5.11 in which the degree of node 1 in the processor graph is greater than that of node 5 (i.e., $d(1) = 4 > d(5) = 3$), while $D_1(1) = 225 < D_1(5) = 289$.

R4. Consider the two encodings of a task tree in Figures 5.12a and 5.12b. G_{T_1} and G_{T_2} in Figure 5.13a and 5.13b are respectively the TR_6's corresponding to the encodings in Figure 5.12a and 5.12b. We then have $N(G_{T_1}, S_4) = 640 > N(G_{T_2}, S_4) = 616$ and $N(G_{T_1}, P_4) = 482 < N(G_{T_2}, P_4) = 484$ where S_4 and P_4 are respectively the star and path with 4 nodes. It can be verified that the encoding of the task tree in Figure 5.12a is the best encoding when $G_P = P_4$, and on the other hand the encoding in Figure 5.12b is the best encoding when $G_P = S_4$. This fact indicates that the best encoding of a task tree depends on the structure of the associated processor graph, i.e., $N(G_{T_1}, G_{P_1}) > N(G_{T_2}, G_{P_1})$ does not always imply $N(G_{T_1}, G_{P_2}) > N(G_{T_2}, G_{P_2})$ for $G_{P_2} \neq G_{P_1}$.

As can be seen from the above remarks, the task assignment problem is more complicated than it may appear to be. This is the very reason that a rigorous procedure like the one treated in this chapter must be called for.

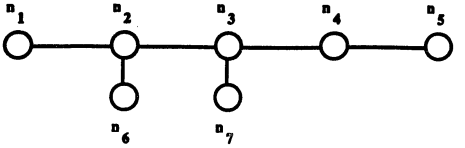


(a) Processor graph

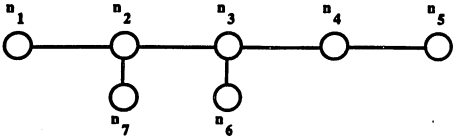


(b) Task graph

Figure 5.11. An example network for Remark 3 where $D_1(1) < D_1(5)$ and $d(1)=4 > d(5)=3$.

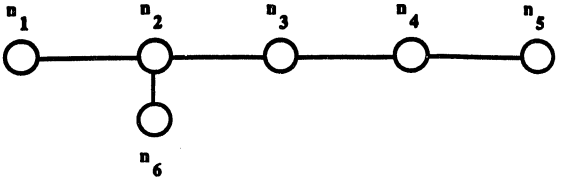


(a) An encoding of a task tree

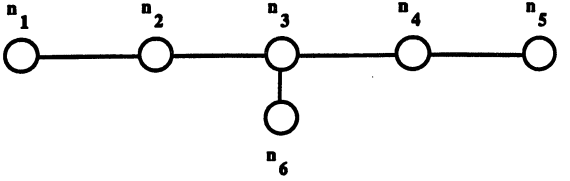


(b) Another encoding of a task tree

Figure 5.12. Example of different encodings which are associated with different enumeration trees.



(a) C_{T_1}, TR_6 according to Figure 12(a)



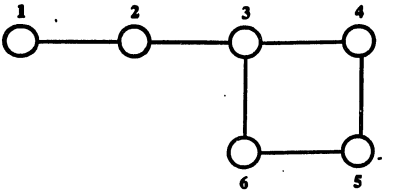
(b) C_{T_2}, TR_6 according to Figure 12(b)

Figure 5.13. TR_6 according to the encoding in Figure 5.12.

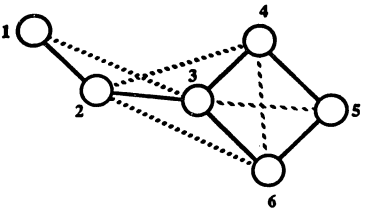
5.2.3. Extension

Thus far, we dealt with only those task assignments with dilations not greater than one. However, our results developed for the case of the dilation not greater than one can be extended to to the completely general case, in which the dilation can be greater than one. Suppose the allowable dilation (AD) is a positive integer $k > 1$. Then, a *communication graph* G_C can be obtained from the processor graph G_P in such a way that $V_C = V_P$ and every pair of processor nodes in G_C is connected iff the number of hops between the pair of processor nodes in G_P is less than or equal to k. For example, given the processor graph in Figure 5.14a and AD = 2, we have the communication graph in Figure 5.14b where a solid line means a one-hop communication, and a dashed line denotes a two-hop communication. Clearly, when AD = 1, the communication graph is the same as the processor graph.

Notice that the constraint "every two related task modules in G_T must be assigned to either a single processor or two processors, the distance between which is less than or equal to k in G_P" is equivalent to the constraint "every two related task modules in G_T must be assigned to either a single processor or two adjacent processors in G_C". Then, we can treat the task assignment problem with processor graph G_P and AD > 1 as the task assignment problem with processor graph G_C and AD = 1. By substituting the communication graph G_C for the processor graph G_P in our previous results, we can formulate $N(G_T, G_C)$, instead of $N(G_T, G_P)$. Following exactly the same procedures in previous examples, we can apply this knowledge to provide a simplified state-space search (as in Example 5.2) and reduce the size of the state-space to be searched (as in Example 5.3).



(a) Processor graph G_p



(b) Communication graph G_c

Figure 5.14. Determination of a communication graph.

CHAPTER 6

TASK ALLOCATION AND MIGRATION IN HYPERCUBE MULTICOMPUTERS

In a hypercube multicomputer, an incoming task must be assigned to a subcube of the required size for execution. Upon completion of execution, the subcube used for the task will be released (relinquished or deallocated) for later use. Efficient allocation and/or deallocation of node processors in a hypercube multicomputer is a key to its performance and utilization. The processor allocation in a hypercube multicomputer consists of two steps: (i) *determination* of the size of a subcube to accommodate an incoming task, and (ii) *location* of a subcube of the size determined by (i) within the hypercube multicomputer. The first step is usually done manually by the users. When the incoming task is specified in a graph form, the first step can be resolved in light of various results developed for the graph embedding [40, 29, 35, 39, 12]. Developing efficient strategies for the second step is the subject of this chapter. Some task allocation strategies have been developed in [24, 22], and some results on the existence of subcubes of certain dimensions in an injured hypercube have been reported [4]. However, we shall address on the problem of searching available subcubes by a sequential search on the addresses of hypercube nodes, thereby distinguishing this work from those described in [4, 12, 22, 24]. Notice that there is a close resemblance of the processor allocation problem in the n-cube multicomputer to the conventional memory allocation problem. In both problems, we want to maximize the utilization of available resources and also minimize the inherent system fragmentation.

Some properties of the buddy strategy will be explored first. Then, the buddy strategy will be proven to be *statically optimal* in the sense that only minimal subcubes are used by the strategy to accommodate each sequence of incoming requests when processor relinquishment is not considered, i.e., static allocation. However, in the case when processor relinquishment is taken into account, the buddy strategy will be shown to be poor in *recognizing* or *detecting* the availability of subcubes in the n-cube multicomputer. Due to the special structure of the n-cube multicomputer, the availability of some subcubes cannot be detected by the buddy strategy, and the processor utilization is thus degraded. The ability of detecting the availability of subcubes will henceforth be termed as *subcube recognition ability*.

A processor allocation strategy using the binary reflected Gray code, the GC strategy, is then described. The performances of both the buddy and GC strategies will be comparatively analyzed. We shall show that the GC strategy, as well as the buddy strategy, is optimal for the static allocation. Furthermore, it will be proven that subcube recognition ability is enhanced significantly by the GC strategy; the number of recognizable subcubes in the GC strategy is twice that in the buddy strategy. We also prove that the subcube recognition ability of the GC strategy is optimal among those achievable sequential searchs. As will be seen, there are many different GC's for an n-cube multicomputer, each of which is associated with a set of recognizable subcubes. An allocation strategy using more than one GC can usually recognize a greater number of available subcubes (and thus improve processor utilization) than can a strategy using only one GC. The relationship between the GC's employed and their subcube recognition ability will also be derived and used for determining an allocation strategy with multiple GC's.

However, like the conventional memory management, allocation and deallocation of subcubes usually result in a fragmented hypercube, i.e., even if a sufficient number of nodes are

available, they do not form a subcube large enough to accommodate an incoming task. Figure 6.1 shows an example of a fragmented hypercube where 4 available nodes cannot form a 2-cube or Q_2. Thus, if a task requiring a Q_2 arrives, it has to be either queued or rejected. As it will be seen later, such fragmentation leads to poor utilization of hypercube nodes, and the improvement achieved by the GC strategy is thus limited. As the fragmentation problem in the conventional memory allocation can be handled by memory compaction, the fragmentation problem in a hypercube can be solved by *task migration*, i.e., relocating and compacting active tasks within the hypercube to one end so as to make large subcubes available in the other end. Notice that due to the special feature of hypercube structures, there is a strong dependence of task migration on the subcube allocation strategy used, since active tasks must be relocated in such a way that the availability of subcubes can be detected by that allocation strategy.

Consequently, we also develop in this chapter a procedure for task migration under the GC strategy. A collection of occupied subcubes is called a *configuration*. The goal configuration to which a given fragmented configuration must change by relocating active tasks is determined first. A task is allocated to a subcube, at each node of which a portion of the task called a *task module* is located. The action for a hypercube node to move its task module to one of its neighboring nodes is called a *moving step*. The cost of each task migration is then measured in terms of the number of required moving steps while task migrations between different pairs of source and destination subcubes are allowed to be performed *in parallel*. Note that to move tasks in parallel, it is very important to avoid deadlocks during the task migration. We not only formulate the node-mapping between each pair of source and destination subcubes so that the number of required moving steps is minimized, but also develop a routing procedure to follow shortest deadlock-free paths for task migration.

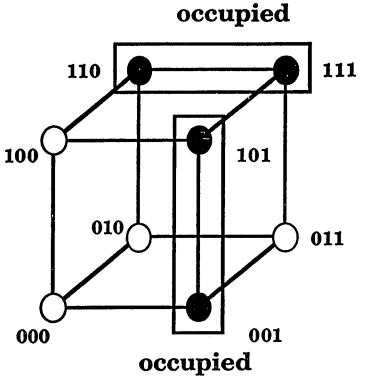


Figure 6.1. A fragmented hypercube.

The chapter is organized as follows. A detailed description of the buddy and GC strategies is given in Section 6.1. Section 6.2 deals with the relationship between the number of GC's employed in an allocation strategy and the corresponding subcube recognition ability. Also given are illustrative examples and simulation results. In Section 6.3, we develop a procedure for task migration under the GC strategy. The goal configuration, the node-mapping between the source and destination subcubes, and a routing scheme which leads to the shortest deadlock-free paths for task migration will be developed in three subsections, respectively. The results presented in this chapter have also been reported in [13, 16].

6.1. Subcube Allocation Strategy Using the Binary Reflected Gray Code

The notation and definitions required will be given in Section 6.1.1. The buddy and GC strategies are introduced in Section 6.1.2 and the subcube recognition ability of the GC strategy is explored in Section 6.1.3.

6.1.1. Preliminaries

The notation and definitions given in Section 4.1.1 will be followed in this chapter. Let $\{g_1, g_2, \cdots, g_n\}$ be a permutation of $\{1, 2, \cdots, n\}$. For $1 \le i \le n$ the *partial rank* r_i of g_i is defined as the rank of g_i in the set $\{g_1, g_2, \cdots, g_i\}$ when the set is rearranged in ascending order. For example, when $\{g_1, g_2, g_3\} = \{3, 1, 2\}$, $r_1 = 1$, $r_2 = 1$ and $r_3 = 2$. Let A be a sequence of binary strings of length n–1, n>1. Then, a sequence of binary strings of length n, denoted by A^{bk}, $b \in \{0, 1\}$, can be obtained by either inserting a bit b into the position immediately right of the bit in the k-th position of every string in A if $1 \le k \le n-1$, or prefixing a bit b to every string in A if $k = n$. Also, let A^* denote the sequence of binary strings obtained from A by reversing the order of the strings in A. For example, if A = {00,

01, 11, 10}, we have $A^{\Gamma 2}$ ={010, 011, 111, 110}, $A^{\Gamma 3}$ = {100, 101, 111, 110}, and A^{*} = {10, 11, 01, 00}. Using the above notation, Gray codes are defined formally as follows.

Definition 6.1: Let G_n be the GC with parameters g_i, $1 \le i \le n$, where $\{g_1, g_2, \cdots, g_n\}$ is a permutation of $Z_n \equiv \{1, 2, \cdots, n\}$. Then, G_n is defined recursively as follows.

$G_1 = \{0, 1\}$,

$G_k = \{G_{k-1}^{0^{r_k}}, (G_{k-1}^{*})^{\Gamma^{r_k}}\}$, $\quad 2 \le k \le n$,

where r_k, as before, is the partial rank of g_k.

For example, for a given GC with parameters $\{g_1, g_2, g_3\} = \{2, 3, 1\}$, we get $\{r_1, r_2, r_3\} = \{1, 2, 1\}$, G_1={0, 1}, G_2={$\underline{0}0$, $\underline{0}1$, $\underline{1}1$, $\underline{1}0$} and G_3={$00\underline{0}$, $01\underline{0}$, $11\underline{0}$, $10\underline{0}$, $00\underline{1}$, $01\underline{1}$, $11\underline{1}$, $10\underline{1}$}, where the newly inserted bits are underlined. It is worth mentioning that the above definition is a generalization of the Gray codes commonly encountered in the literature [11], and that the *binary reflected Gray code* (BRGC), the most frequently used GC, can be obtained readily from this definition by letting g_i=i, $1 \le i \le n$. Figure 6.2a and 6.2b show respectively the BRGC and a GC with $\{g_1, g_2, g_3\} = \{2,3,1\}$. Note that a GC with parameters g_i, i = 1,...,n, can be obtained by permuting the bits in the BRGC in such a way that the direction i of the BRGC becomes the direction g_i of this GC. For simplicity, unless specified otherwise, G_n will denote the BRGC.

A set of contiguous integers is called a *region* and let #[a, b] = {k | $a \le k \le b$, $k \in I^{+}$}, where I^{+} denotes the set of positive integers. A *coding scheme* with n bits, denoted by C_n, is defined as a one-to-one mapping from a number within #[0, 2^n–1] to a binary representation with n bits, and let $C_n(m)$ denote the representation of a number m under the coding scheme C_n. A coding scheme with 4 bits which is neither the BRGC nor the standard binary encoding scheme is given in Figure 6.3. Let $B_n(m)$ denote the binary representation of an integer m

000	000
001	010
011	110
010	100
110	101
111	111
101	011
100	001

(a) A BRGC (b) a GC with $\{g_1, g_2, g_3\}=\{2,3,1\}$

Figure 6.2. Illustration of Gray codes.

0. 0000
1. 0001
2. 0011
3. 0010
4. 0110
5. 0111
6. 1111
7. 1110
8. 1100
9. 1000
10. 1010
11. 1011
12. 1001
13. 1101
14. 0101
15. 0100

Figure 6.3. A coding scheme with 4 bits.

with n bits and $G_n(m)$ be the BRGC representation of m. Also, the notation $\lfloor b \rfloor$ is used to denote the largest integer which is less than or equal to b, and $\lceil b \rceil$ the smallest integer which is greater than or equal to b. Let $|S|$ denote the cardinality of the set S.

6.1.2. Subcube allocation strategies

Node processors in an n-cube multicomputer must be allocated to incoming tasks so as to maximize processor utilization and minimize system fragmentation. Due to the special structure of an n-cube, it is very difficult to *detect* the availability of a subcube of required size, and *merge* released subcubes of small sizes into a larger subcube. Although the procedures used in determining the primary implicants in a Boolean function can be used in detecting the availability of a subcube, they are too complicated to be useful [51]. Thus, we shall not follow these procedures; instead, we shall focus on ways of exploiting the ideas used in conventional memory allocation approaches. An allocation strategy using the buddy system, the buddy strategy, will be described first. Next, a strategy using the BRGC, the GC strategy, will be described. The GC strategy will be shown to outperform the buddy strategy. As will be seen later, our approach to the processor allocation problem is mainly based on a sequential search of a list of allocation bits, and can thus be accomplished by the various approaches used for the conventional memory allocation, such as the first-fit and the best-fit searches [81]. For clarity of presentation, the first-fit approach is employed in both the buddy and GC strategies.

6.1.2.1. The buddy strategy

One form of the buddy strategy was investigated in [67] and also implemented in the NCUBE multicomputer [20]. Since there are 2^n processor nodes in a Q_n, 2^n allocation bits are

used to keep track of the availability of all the nodes. An allocation bit with value 0 (1) indicates the availability (unavailability) of the corresponding node. The buddy strategy consists of two parts, processor *allocation* and processor *relinquishment*, which are outlined below.

Processor Allocation

Step 1. Set k := $\lceil I_j \rceil$, where $\lceil I_j \rceil$ is the dimension of a subcube required to accommodate the request I_j.

Step 2. Determine the least integer m such that all the allocation bits in the region $\#[m2^k, (m+1)2^k-1]$ are 0's, and set all the allocation bits in the region $\#[m2^k, (m+1)2^k-1]$ to 1's.

Step 3. Allocate nodes with addresses $B_n(i)$ to the request I_j, $\forall\ i \in \#[m2^k, (m+1)2^k-1]$.

Processor Relinquishment

Reset every p-th allocation bit to 0, where $B_n(p) \in q$ and q is the address of a released subcube.

This strategy can be explained by the binary tree in Figure 6.4. The level where the root node resides is numbered 0, and the nodes in level i are associated with subcubes of dimension n–i. A node in this binary tree is available only if all of its offsprings are available. When an incoming request needs a Q_k, the buddy strategy searches the level n–k of the tree from left to right and allocates the first available subcube to the request. The processors associated with allocation bits in $\#[m2^k, (m+1)2^k-1]$ always constitute a Q_k whose address is $B_{n-k}(m)^{*k}$. Similarly to the conventional memory allocation, whenever a processor allocation or relinquishment takes place, the subcube to be allocated or released must be associated with a region of contiguous allocation bits.

Static allocation is concerned only with how to accommodate incoming requests without considering processor relinquishment. Figure 6.5 shows a simple example of static allocation.

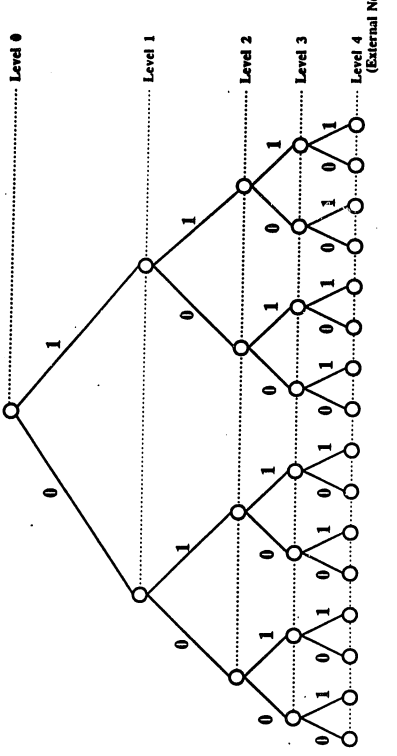


Figure 6.4. A complete binary tree for the buddy strategy.

$I_1 = Q_0$ $I_5 = Q_1$
$I_2 = Q_2$ $I_6 = Q_2$
$I_3 = Q_0$ $I_7 = Q_0$
$I_4 = Q_0$ $I_8 = Q_1$

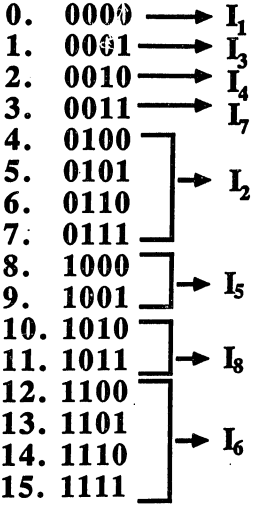


Figure 6.5. The operations of the buddy strategy.

It is easy to observe that a Q_4 can accommodate the incoming request sequence $\{I_1, \cdots, I_8\}$ even if the order of the requests in the sequence were arbitrarily shuffled. As we shall prove later, this is not a coincidence, but rather, a result of the *static optimality*. An allocation strategy is said to be *statically optimal* if a Q_n using the strategy can accommodate any input request sequence $\{I_i\}_{i=1}^{k}$ iff $\sum_{i=1}^{k} 2^{|I_i|} \leq 2^n$, where $|I_i|$ is the dimension of a subcube required to accommodate the request I_i. Following the concept of the buddy system described in [49, 67], we have the proposition below for the buddy strategy.

Proposition 6.1: The buddy strategy is statically optimal.

Note that when a Q_k is needed, the buddy strategy searches for a region of allocation bits with 0's whose addresses start with an integral multiple of 2^k. This in turn implies that there are only 2^{n-k} Q_k's within the n-cube multicomputer recognizable by the buddy strategy. Consequently, the buddy strategy under-utilizes processors in the n-cube multicomputer. To remedy the processor under-utilization problem of the buddy strategy, we shall describe a allocation strategy using the BRGC. Under the GC strategy, processor utilization will be improved significantly by novel mappings between the allocation bits and node processors.

6.1.2.2. The GC strategy

Similarly to the buddy strategy, the GC strategy can be described by the following two parts.

Processor Allocation

Step 1: Set $k := |I_j|$, where $|I_j|$ is the dimension of a subcube required to accommodate the request I_j.

Step 2: Determine the least integer m such that all (i mod 2^n)-th allocation bits are 0's, where $i \in \#[m2^{k-1}, (m+2)2^{k-1}-1]$. Set all these 2^k allocation bits to 1's.

Step 3: Allocate nodes with addresses $G_n(i \bmod 2^n)$ to I_j, where $i \in \#[m2^{k-1}, (m+2)2^{k-1}-1]$.

Processor Relinquishment

Reset every p-th allocation bit to 0, where $G_n(p) \in q$, and q is the address of a subcube released.

Since the nodes corresponding to the first and last allocation bits are adjacent to each other, a circular search is allowed in the GC strategy. To show that the nodes associated with those allocation bits in $\#[m2^{k-1}, (m+2)2^{k-1}-1]$ constitute a Q_k, consider another procedure for generating the BRGC [11]. (As mentioned before, one can assume, without loss of generality, that the GC strategy uses the BRGC only.) It is proved in [70] that this procedure indeed generates the BRGC. Given a k-bit BRGC $G_k = \{d_0, d_1, \cdots, d_{2^k-1}\}$, a (k+1)-bit BRGC can be generated by:

$$G_{k+1} = \{d_0 0, d_0 1, d_1 1, d_1 0, d_2 0, d_2 1, \cdots, d_{2^k-1} 1, d_{2^k-1} 0\}.$$

This procedure can be described by a complete binary tree as in Figure 6.6. The address of every external node is determined by the coded bits in the path from the root to the external node, and the BRGC is then obtained by the addresses of external nodes from left to right.

Similarly to the binary tree in Figure 6.4, the nodes in level n–k are associated with Q_k's. It is easy to see from the scheme of coding edges of the tree that two adjacent nodes in the (n–k+1)-th level form a Q_k, even when they do not have the same immediate predecessor. Therefore, when a Q_k is requested, the GC strategy searches from left to right for two adjacent available nodes in level n–k+1, rather than searching for an available node in level n–k. A Q_k will thus be searched for in the regions whose addresses start with an integral multiple of 2^{k-1} instead of 2^k. (Recall that the latter was used for the buddy strategy.) This means that

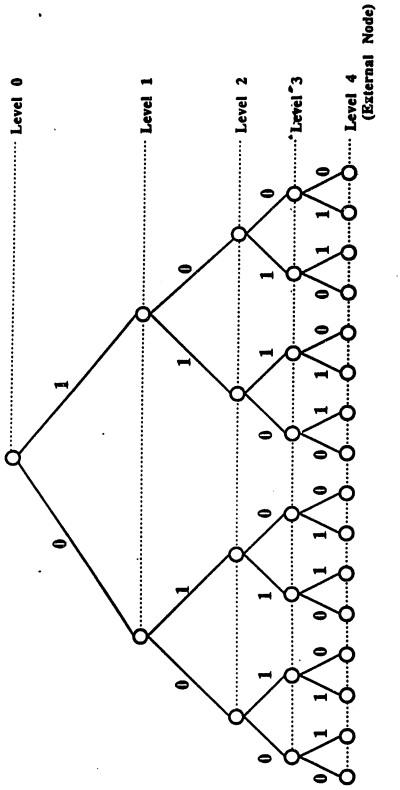


Figure 6.6. A complete binary tree for the GC strategy.

the number of subcubes recognizable by a GC is twice that by the buddy strategy. The number of subcubes recognizable by each of the two strategies is presented in Table 6.1.

Because of its enhanced subcube recognition ability, the GC strategy can allocate subcubes more densely at one end, and, thus, make larger subcubes available at the other end for future use. Moreover, the GC strategy too is statically optimal.

Theorem 6.1: The GC strategy is statically optimal.

Proof: In Step 2 of the processor allocation part of the GC strategy, a request is said to be *type A* (*type B*) if the request is allocated to those nodes with addresses $G_n(i \bmod 2^n)$, $i \in \#[m2^{k-1}, (m+2)2^{k-1}-1]$, where m is an even (odd) number. Note that whether an incoming request is type A or type B depends on the allocation bits set by previous requests. If a request is type A, both the buddy and GC strategies can allocate a subcube to the request while setting the same region of allocation bits to 1's. (For type A requests there is no distinction between the buddy and GC strategies.) However, if I_p is a type B request and $\#[m2^{|I_p|-1}, (m+2)2^{|I_p|-1}-1]$ is its corresponding region, then we claim that there is no region forming a $Q_{|I_p|-1}$ before the region $\#[m2^{|I_p|-1}, (m+2)2^{|I_p|-1}-1]$. This claim implies that every type B request I_p can be decomposed into two contiguous requests, each asking for a subcube of dimension $|I_p|-1$, and these two requests can be allocated within the n-cube under the buddy strategy while still setting the same region of allocation bits, $\#[m2^{|I_p|-1}, (m+2)2^{|I_p|-1}-1]$, to 1's.

To prove the claim, suppose I_p is a type B request and $\#[m2^{|I_p|-1}, (m+2)2^{|I_p|-1}-1]$ is the corresponding region to be allocated. Then, the region $\#[(m-1)2^{|I_p|-1}, m2^{|I_p|-1}-1]$ must have become unavailable as a consequence of allocating those requests asking for (1) subcubes of dimension less than or equal to $|I_p| - 1$, or (2) subcubes of dimension $|I_p|$. However, without loss of generality, I_p can be assumed to be the first type B request asking for a $Q_{|I_p|}$,

	Q_0	Q_k, $1 \leq k \leq n-1$	Q_n
The number of distinct subcubes	2^n	$C_k^n 2^{n-k}$	1
The number of distinct subcubes recognizable by the buddy system	2^n	2^{n-k}	1
The number of distinct subcubes recognizable by a GC	2^n	2^{n-k+1}	1

Table 6.1. The number of subcubes recognizable by the buddy and GC strategies.

and case (2) can thus be ignored. Then, there is no no region forming a $Q_{|I_p|-1}$ before $\#[m2^{|I_p|-1}, (m+2)2^{|I_p|-1}-1]$. The claim is thus proved.

For any incoming request sequence $\{I_1, I_2, \cdots, I_j\}$, every type B request in $\{I_1, I_2, \cdots, I_{j-1}\}$ is decomposed into two contiguous requests as mentioned above and the buddy strategy is applied to allocate subcubes to the resulting request sequence. The availability of a region large enough to accommodate I_j is guaranteed by the static optimality of the buddy strategy, as stated in Proposition 6.1. **Q.E.D.**

An example of the GC strategy is given in Figure 6.7, where the input request sequence is the same as that in Figure 6.5. It can be observed that the GC strategy outperforms the buddy strategy in the first-fit search and will "pack" incoming requests more densely, thus making larger contiguous regions available than the buddy strategy can.

6.1.3. Subcube recognition ability of a single GC

Let $\{g_1, g_2, \cdots, g_n\}$ be a permutation of Z_n. As mentioned earlier, by permuting the i-th direction of the BRGC to the g_i-th direction, one can obtain a GC with parameters g_i, i = 1,...,n. Since there are n! permutations of n distinct numbers, there are n! distinct GC's for the n-cube multicomputer. Moreover, the subcube recognition ability of each GC can be determined by the following theorem.

Theorem 6.2: A subcube Q_k with the address $b_n b_{n-1} \cdots b_1$ can be recognized by a GC with parameters g_i, $1 \le i \le n$, iff any of the following three conditions is satisfied:

(i) $b_{g_i} = *$, $1 \le i \le k$.

(ii) $b_{g_i} = *$, $1 \le i \le k-1$, and there exists an r such that $b_{g_{r-1}}=1$, $b_{g_r}=*$ and $b_{g_s}=0$, $k \le s < r-1$.

$I_1 = Q_0$ $\quad I_5 = Q_1$

$I_1 = Q_2$ $\quad I_6 = Q_2$

$I_3 = Q_0$ $\quad I_7 = Q_0$

$I_4 = Q_0$ $\quad I_8 = Q_1$

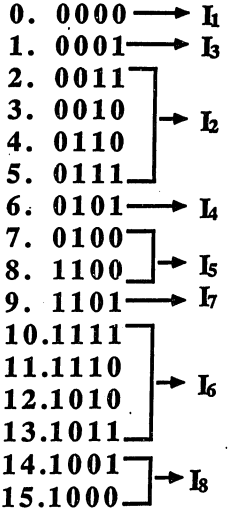


Figure 6.7. The operations of the GC strategy.

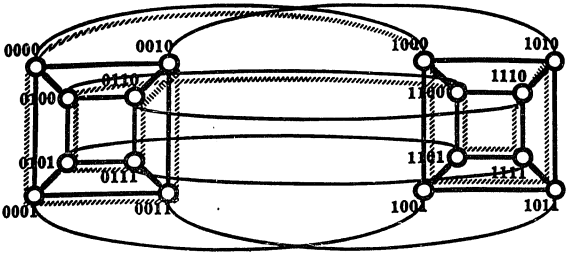
(iii) $b_{g_i} = *$, $1 \le i \le k-1$, $b_{g_s} = 0$, $k \le s \le n-1$, and $b_{g_n} = *$.

Proof: Without loss of generality, one can consider the BRGC only, since all the other cases can be obtained by permuting the subcube address bits. Consider the complete binary tree in Figure 6.6. Condition (i) is associated with all the nodes in level n–k, i.e., Q_k's. There are 2^{n-k} such Q_k's. Condition (ii) corresponds to the availability of the two adjacent nodes in level n–k+1 whose youngest common ancestor is in level n–r, $k+1 \le r \le n$. The number of such Q_k's is $\sum_{i=0}^{n-k-1} 2^i$. Condition (iii) is associated with the Q_k that results from the two nodes at both the ends of level n–k+1. Thus, the total number of recognizable Q_k's is $2^{n-k} + \sum_{i=0}^{n-k-1} 2^i + 1 = 2^{n-k+1}$, which agrees with Table 6.1. **Q.E.D.**

An illustrative example of the subcube recognition ability of a 4-bit BRGC is shown in Figure 6.8. It is interesting to observe that the node addresses of a subcube recognizable by a GC are contiguous in that GC. This is the very reason that a GC can be used to detect the availability of subcubes with the sequential search designed for the conventional memory allocation. Moreover, as it will be proved below, the GC strategy is optimal insofar as the subcube recognition ability of a sequential search is concerned.

Theorem 6.3: The BRGC is an optimal coding scheme as far as the subcube recognition ability of a sequential search is concerned.

Proof: Note that there are 2^{n-k+1} Q_k's recognizable by a BRGC [13]. Suppose there exists a coding scheme that can recognize more than 2^{n-k+1} Q_k's. Then, there must exist two distinct integers i and j such that (a) $|i-j| < 2^{k-1}$, (b) nodes with addresses $C_n(p)$, $p \in \#[i, i+2^k-1]$, form a Q_k, and (c) nodes with addresses $C_n(q)$, $q \in \#[j, j+2^k-1]$, form another Q_k. Thus, the cardinality of $\#[i, i+2^k-1] \cap \#[j, j+2^k-1]$ must be greater than 2^{k-1} since $|i-j| <$



Recognizable subcube types			Total number of recognizable subcubes
Q_0 :	dddd		16
Q_1 :	ddd*	(i)	8 + 4 + 2 + 1 + 1 = 16
	dd*1 d*10 *100	(ii)	
	*000	(iii)	
Q_2 :	dd**	(i)	4 + 2 + 1 + 1 = 8
	d*1* *10*	(ii)	
	00	(iii)	
Q_3 :	d***	(i)	2 + 1 + 1 = 4
	*1**	(ii)	
	*0**	(iii)	

Figure 6.8. Illustration of Theorem 6.3 when $\{g_1, g_2, g_3, g_4\}$ where d is 0 or 1.

2^{k-1}, and less than 2^k since $i \neq j$. However, this means that the intersection of two subcubes does not form a subcube, leading to a contradiction. **Q.E.D.**

6.2. Allocation Strategies Using Multiple GC's

Although the GC strategy has better subcube recognition ability than the buddy strategy, it still cannot recognize all the subcubes in an n-cube multicomputer. Note that different GC's are associated with different sets of recognizable subcubes. Thus far, we have considered the case of using a single GC only. Clearly, subcube recognition ability will improve if more than one GC is used. It is therefore important to investigate the relationship between the number of GC's employed and the corresponding subcube recognition ability.

Note that Theorem 6.2 provides a necessary and sufficient condition for the availability of a subcube to be recognized by a GC. Since different GC's are associated with different sets of recognizable subcubes, processor utilization in an n-cube multicomputer improves as the number of GC's used in an allocation strategy increases. Consider an allocation strategy which uses three GC's with the following parameters: $\{g_j^1\}_{j=1}^5 = \{1, 2, 3, 4, 5\}$, $\{g_j^2\}_{j=1}^5 = \{2, 5, 1, 3, 4\}$ and $\{g_j^3\}_{j=1}^5 = \{3, 1, 4, 5, 2\}$. Then, the set of subcubes recognizable by this allocation strategy can be determined by Theorem 6.2 and shown in Figure 6.9 with the trivial cases for Q_0 and Q_5 omitted. Also, it is shown from Theorem 6.2 that every subcube must be recognizable by at least one GC and that the complete subcube recognition can be achieved if all the n! GC's are used. However, we naturally want to reduce, if possible, the number of GC's required for complete subcube recognition in order to minimize the search overhead associated with multiple GC's. More on this will be discussed in the following subsection.

GC ={1,2,3,4,5}	GC ={2,5,1,3,4}	GC ={3,1,4,5,2}
dddd*	ddd*d	dd*dd
ddd*1	*dd1d	dd1d*
dd*10	1dd0*	d*0d1
d*100	0d*01	*10d0
*d000	0*d00	d00*0
ddd**	*dd*d	dd*d*
dd*1*	1dd**	d**d1
d*10*	0d**1	*1*d0
d00	0*d*0	d0**0
dd***	*dd**	d**d*
d*1**	*d**1	*1*d*
*d0**	**d*0	d0***
d****	*d***	***d*
*d***	**d**	d****

Figure 6.9. Recognizable subcubes by the given Gray codes, $\{g_1, g_2, g_3, g_4, g_5\}=\{1,2,3,4,5\}$, $\{2,5,1,3,4\}$, $\{3,1,4,5,2\}$ where $d \in \{0,1\}$.

6.2.1. The number of GC's for complete subcube recognition

Let S be a set of strings which are permutations of Z_n. S is said to have the *C-property* if for any k distinct numbers from Z_n there is at least one string $s \in S$ such that these k numbers are the first k numbers of s.

Lemma 6.1: Let M_k be the set of all combinations of k distinct integers out of Z_n, $0 \le k \le n$, and let $x \overset{\bullet}{\leq} y$ denote that all the integers in a combination x are contained in another combination y. Then,

(i) There is a one-to-one function $f: M_i \rightarrow M_{i+1}$, $0 \le i \le \lfloor \frac{n}{2} \rfloor - 1$, such that

$\forall\, x \in M_i$, $x \overset{\bullet}{\leq} f(x)$, and

(ii) There is a one-to-one function $g: M_{i+1} \rightarrow M_i$, $\lfloor \frac{n}{2} \rfloor \le i \le n-1$, such that

$g(x) \overset{\bullet}{\leq} x$.

It is necessary to introduce the Theorem of Matching in [54] to prove Lemma 6.1.

Theorem of Matching [54]: In a bipartite graph G=(V,E), a complete matching from $X \subseteq V$ to $Y \subseteq V$ exists iff $|A| \le |R(A)|$ for every subset A of X, where R(A) denotes the set of vertices in Y that are adjacent to vertices in A.

A matching in a bipartite graph is a selection of edges such that no two edges are incident with the same vertex, and a complete matching from X to Y in a bipartite graph is a matching in which there is an edge incident with every vertex in X. For example, there is a complete matching from X to Y in Figure 6.10a, but not in Figure 6.10b, since $|\{x_2, x_3\}| = 2 > |R(\{x_2, x_3\})| = |\{y_4\}| = 1$ in Figure 6.10b. Lemma 6.1 can now be proved as follows.

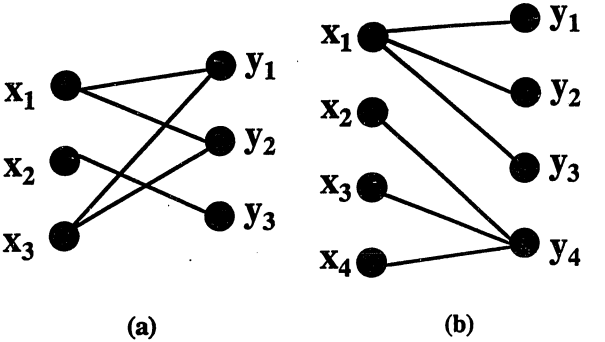


Figure 6.10. Example for illustrating Theorem of Matching.

Proof of Lemma 6.1: Consider (i) first. Draw an edge between $x \in M_i$ and $y \in M_{i+1}$ if $x \overset{*}{\leq} y$. An example is given in Figure 6.11a for n=5 and i=1. Note that every element in M_i is connected to n–i elements in M_{i+1}, and every element in M_{i+1} is connected to i+1 elements in M_i. Treat M_i and M_{i+1} as X and Y, respectively in the Theorem of Matching. Then, for each subset D of M_i, there are $|D|(n-i)$ edges incident with the elements in D. Since $n-i > i+1$, these edges must be incident with $|D|$ or more elements in M_{i+1}. The condition corresponding to $|A| \leq |R(A)|$ in the Theorem of Matching is satisfied, and, thus, (i) follows. By treating M_{i+1} as X and M_i as Y, (ii) can be proved similarly. **Q.E.D.**

Figure 6.11b illustrates Lemma 6.1 when n=5 and i=1.

Theorem 6.4: Let SC be the set of all sets with the C-property. Then, $\min_{S \in SC} \{|S|\} = C^n_{\lfloor \frac{n}{2} \rfloor}$, where "C" stands for "combination."

Proof: Since for every S with the C-property, $|S| \geq C^n_k$, $\forall\ k \in Z_n \cup \{0\}$. Clearly, $\min_{S \in SC} \{|S|\} \geq \max_{k \in Z_n \cup \{0\}} C^n_k = C^n_{\lfloor \frac{n}{2} \rfloor}$. The equality $\min_{S \in SC} \{|S|\} = C^n_{\lfloor \frac{n}{2} \rfloor}$ is proved below by showing the existence of a set with the C-property whose cardinality is $C^n_{\lfloor \frac{n}{2} \rfloor}$.

Construct a set of strings with the C-property as follows. Determine the one to one function f from M_i to M_{i+1} for every pair of sets M_i and M_{i+1}, $0 \leq i \leq \lfloor \frac{n}{2} \rfloor - 1$, and draw an arc from f(x) to x. Also, determine the one to one function g from M_{i+1} to M_i for every pair of sets M_{i+1} and M_i, $\lfloor \frac{n}{2} \rfloor \leq i \leq n-1$, and draw an arc from x to g(x). (The existence of such functions was proved in Lemma 6.1.) Then, treat every entry in M_i as a string with the i numbers in that entry. For example, {1, 3} of M_2 in Figure 6.11b is treated as 13 or 31. Next, the strings in M_i are determined in such a way that if there is an arc from a string $s \in M_i$ to a string $t \in M_{i-1}$, then t is a permutation of the first i–1 integers of s. Note that this

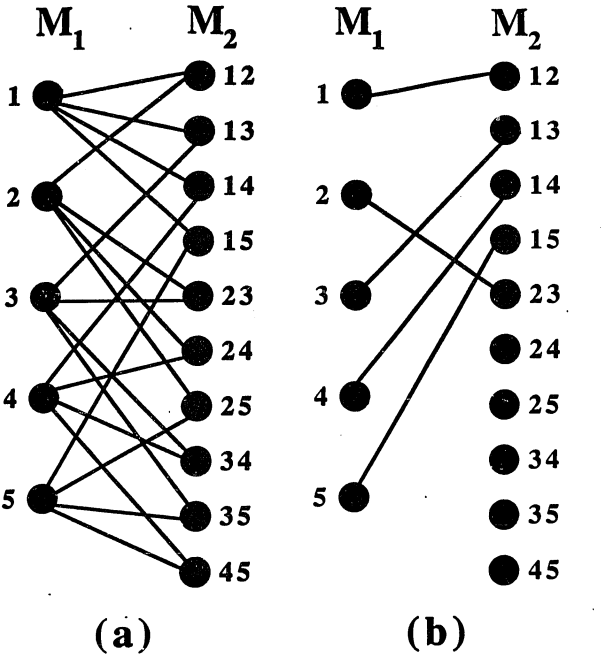


Figure 6.11. A complete matching from M_1 to M_2.

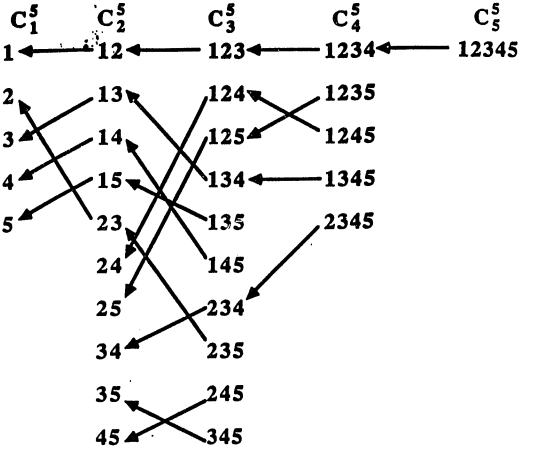
procedure can be performed from M_n to M_1 step by step and a later permutation of leading integers does not affect what the string was used for in previous steps. From Lemma 6.1, there are $C^n_{\lfloor \frac{n}{2} \rfloor}$ strings without incoming arcs in M_i, $\lfloor \frac{n}{2} \rfloor \leq i \leq n$. Then, treat every string in M_i as a permutation of Z_n starting with the i numbers. For example, 452 in a M_3 can be 45213 or 45231. From (i) of Theorem 6.2, those $C^n_{\lfloor \frac{n}{2} \rfloor}$ strings after the above modification are the parameters of the GC's required for complete subcube recognition, and, thus, this theorem follows. **Q.E.D.**

An example of determining the GC's required for complete subcube recognition in a Q_5 is given in Figure 6.12. The method introduced in the proof of Theorem 6.4 has placed arcs from M_i to M_{i-1}, $2 \leq i \leq n$, in Figure 6.12a, and the procedure of determining the required GC's is shown in Figure 6.12b.

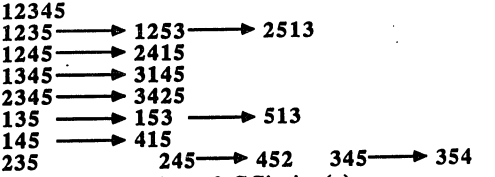
Let $\alpha(n)$ be the minimal number of GC's required for complete subcube recognition in a Q_n. Then, the following corollary follows from Theorem 6.4.

Corollary 6.4.1: $\alpha(n) \leq C^n_{\lfloor \frac{n}{2} \rfloor}$.

Notice that as pointed out in [16], the approach of permuting the bits in the BRGC to obtain other coding schemes with different set of recognizable subcubes can also be applied to the standard binary encoding scheme. More specifically, we can obtain *extended binary coding* (EBC) schemes by permuting the bits in the standard binary encoding scheme, and apply (i) of Theorem 6.2 to determine the recognizable subcubes of each EBC. Let $\beta(n)$ be the minimal number of EBC's required for complete subcube recognition in a Q_n. From Theorem 6.4 and the fact that there are $2^{n-k}C^n_k$ Q_k's in a Q_n and each EBC can only recognize 2^{n-k} Q_k's, we have the following corollary.



(a) Determine the GC's for complete subcube recognition.



(b) Modification of GC's in (a)

Figure 6.12. The GC's required for complete subcube recognition in a Q_5.

Corollary 6.4.2: $\beta(n) = C^n_{\lfloor \frac{n}{2} \rfloor}$.

To determine the complexity of $C^n_{\lfloor \frac{n}{2} \rfloor}$, consider the following proposition.

Proposition 6.2: $\{\prod_{k=1}^{n}(2k-1)\}/(2^n n!) < (2n+1)^{-1/2}$ $\forall$ $n \in \mathbf{I}^+$.

Proof: This inequality is proved by induction. Clearly, the inequality holds for n=1. Assume $\{\prod_{k=1}^{n}(2k-1)\}/(2^n n!) < (2n+1)^{-1/2}$ for an $n \in \mathbf{I}^+$. Then, $\{\prod_{k=1}^{n+1}(2k-1)\}/(2^{n+1}(n+1)!) = \frac{2n+1}{2n+2}\{\prod_{k=1}^{n}(2k-1)\}/(2^n n!) < \frac{(2n+1)^{1/2}}{2n+2} < \{2(n+1)+1\}^{-1/2}$, which completes the proof.

Q.E.D.

From $C^n_{\lfloor \frac{n}{2} \rfloor} = \prod_{k=1}^{\lceil \frac{n}{2} \rceil} \frac{2k-1}{k} 2^{\lfloor \frac{n}{2} \rfloor}$ and the above proposition, it can be verified that $\lim_{n\to\infty} C^n_{\lfloor \frac{n}{2} \rfloor}/2^n = 0$ and $\lim_{n\to\infty} C^n_{\lfloor \frac{n}{2} \rfloor}/a^n = \infty$ $\forall$ $a \in [0, 2)$, meaning that the complexity of $C^n_{\lfloor \frac{n}{2} \rfloor}$ is still exponential. However, the above result bears practical importance, since the number of the GC's required for complete subcube recognition is significantly reduced according to Corollary 6.4.1; especially, this is true when n is large because $\lim_{n\to\infty} C^n_{\lfloor \frac{n}{2} \rfloor}/2^n = 0$.

6.2.2. Subcube recognition ability for an arbitrary set of GC's

Although complete subcube recognition can be accomplished by using no more than $C^n_{\lfloor \frac{n}{2} \rfloor}$ GC's, we may want to reduce further the number of GC's employed so as to lessen the search overhead induced by those GC's while maintaining the required (perhaps not complete) subcube recognition ability. Therefore, the relationship between the subcube recognition ability and the GC's used has to be explored further.

Clearly, since there is a finite number of subcubes in an n-cube multicomputer, the subcube recognition ability for an arbitrary set of GC's can be determined by enumeration in light of Theorem 6.2. This is, however, undesirable due to the high degree of complexity. Instead of a brute-force enumeration, we shall develop below a systematic procedure which uses the parameters of GC's and determines the number of subcubes recognizable by those GC's.

Let SG = $\{G^1, G^2, \cdots, G^h\}$ be a set of h GC's in a Q_n and g_k^i, $1 \le k \le n$, denote the parameters of G^i. Consider the case of determining the number of Q_l's recognizable by SG first. Define a_j^i, b_j^i and c_j^i to facilitate our presentation as follows. For each $j \in Z_n$, when $g_m^i = j$, let $b_j^i = g_{m-1}^i$ if m>1 and $b_j^i = \varepsilon$ if m=1. That is, b_j^i is the number that is in front of the number j in the parameters of G^i, and $b_j^i = \varepsilon$ represents the case when j is the first parameter of G^i, i.e., $j = g_1^i$. Let $a_j^i = \left\{ g_u^i \mid 1 \le u < m-1 \right\}$ and $c_j^i = \left\{ g_u^i \mid m < u \le n \right\}$. In other words, a_j^i is the set of parameters of G^i before the number b_j^i, and c_j^i the set of parameters of G^i after the number j. Note that b_j^i and c_j^i could be empty in some cases.

Define an n-tuple *resemblance vector* or *R-vector*, $(R_1, R_2, \cdots, R_n)$, for the set SG in such a way that $R_j = \theta$ if there exists an integer $p \ne i$ such that $b_j^i \ne \varepsilon$ and $b_j^i \in a_j^p$, and $R_j = \left| \bigcap_{i=1}^{h} c_j^i \right|$ otherwise. The notation $R_j = \theta$ is used to mean that there does not exist any Q_l with a * in the j-th direction of its address that can be recognized by *every* GC $\in$ SG. It can be verified that the condition for $R_j = \theta$ is a result of (ii) in Theorem 6.2. In addition, for $1 \le k \le n$, let $\delta(k)=1$ if $R_k \ne n-1$ and $\forall\, g_w^i = k$, w= 1 or n, and $\delta(k)=0$ otherwise. The necessity of the function δ results from (iii) of Theorem 6.2. Thus, we get $M^{(1)}(SG) = \sum_{j,\, R_j \ne \theta} 2^{R_j} + \sum_{j=1}^{n} \delta(j)$, the number of Q_l's that can be recognized by *every* GC $\in$ SG. Following the same procedure, one can determine $M^{(1)}(B)$ for any $B \subseteq SG$. Then, using the concept of exclusion and inclu-

sion [54], we can get $N^{(1)}(SG)$, the number of Q_1's recognizable by SG as follows.

Let $s_1 = \sum_{i=1}^{h} M^{(1)}(\{G^i\}) = 2^n h$

$$s_2 = \sum_{\substack{i,j \\ i \neq j}} M^{(1)}(\{G^i, G^j\})$$

$$s_3 = \sum_{\substack{i,j,k \\ i \neq j \neq k}} M^{(1)}(\{G^i, G^j, G^k\})$$

$$\vdots$$

$$s_h = M^{(1)}(SG).$$

Then, the number of Q_1's recognizable by SG is $N^{(1)}(SG) = \sum_{i=1}^{h} s_i(-1)^{i+1}$.

For the recognition of subcubes of dimension higher than 1, the above procedure can be generalized as follows. Consider the determination of $N^{(k)}(SG)$, the number of Q_k's recognizable by SG = $\{G^1, G^2, \cdots, G^h\}$. Again, we derive $M^{(k)}(SG)$, the number of Q_k's recognizable by *every* GC $\in$ SG first. Let $W = \bigcup_{i=1}^{h} \bigcup_{j=1}^{k-1} \{g_j^i\}$. By Theorem 6.2, $M^{(k)}(SG)=0$ if $|W| > k$ and, thus, only the cases $|W| = k-1$ and $|W| = k$ have to be considered. In the case of $|W|=k-1$, i.e., the first k–1 parameters of all GC's in SG are the same, it can be seen from Theorem 6.2 that the procedure of determining $N^{(k)}(SG)$ can degenerate into the case of determining the number of recognizable Q_1's while ignoring the parameters in W.

Without loss of generality, one can consider $|W|=k$ only. Let $g_{y_i}^i$ be the parameter in W but not in $\{g_j^i\}_{j=1}^{k-1}$. Note that such a parameter must be unique. Then, let $b^i = g_{y_i-1}^i$ if $y_i > k$ and $b^i = \epsilon$ otherwise, $a^i = \left\{ g_u^i \mid k \le u < y_i - 1 \right\}$, $c^i = \left\{ g_u^i \mid y_i < u \le n \right\}$, and $R_w = \left| \bigcap_{i=1}^{h} c^i \right|$. In addition, let

$\delta(W) = 1$ if $R_w \neq n-k$ and $y_i = k$ or $n \; \forall \; i \in \{1,..., h\}$, and $\delta(W) = 0$ otherwise. Thus, we have $M^{(k)}(SG)=0$ if there exists an integer $p \neq i$ such that $b^i \neq \varepsilon$ and $b^i \in a^p$, and $M^{(k)}(SG) = 2^{R_w} + \delta(W)$ otherwise. Following the same procedure of exclusion and inclusion as above, $N^{(k)}(SG)$ can be determined.

Using the parameters of GC's, the number of recognizable subcubes of given dimension can therefore be computed systematically by the above procedure. To illustrate this procedure, consider the subcube recognition ability of an allocation strategy using the three GC's in Figure 6.9 for a Q_5. That is, $SG=\{G^1, G^2, G^3\}$, $\{g_j^1\}_{j=1}^5 = \{1, 2, 3, 4, 5\}$, $\{g_j^2\}_{j=1}^5 = \{2, 5, 1, 3, 4\}$ and $\{g_j^3\}_{j=1}^5 = \{3, 1, 4, 5, 2\}$. To determine $M^{(1)}(\{G^1, G^2\})$, we have $a_1^1=\varnothing$, $b_1^1=\varepsilon$, $c_1^1=\{2, 3, 4, 5\}$ and $a_1^2=\{2\}$, $b_1^2=5$, $c_1^2=\{3, 4\} \Rightarrow R_1=2$. Then, $R_2=3$, $R_3=0$, $R_4=0$, $R_5=0$, and $\delta(k)=0$ for $1\leq k\leq 5$, leading to $M^{(1)}(\{G^1, G^2\}) = 2^2 + 2^3+2^0 = 13$. Similarly, we get $M^{(1)}(\{G^1, G^3\}) = 2^3+ 2^2 +2^0 = 13$ ($R_1=3$, $R_2=0$, $R_3=2$, $R_4=0$, $R_5=0$, and $\delta(k)=0$ for $1\leq k\leq 5$), $M^{(1)}(\{G^2, G^3\}) = 2^1+ 2^0+ 2^1 +2^0 + 1 = 7$ ($R_1=1$, $R_2=0$, $R_3=1$, $R_4=0$, $R_5=0$, $\delta(2)=1$ and $\delta(k)=0$ k =1, 3, 4, 5), and $M^{(1)}(SG)=2$ ($R_1=1$, $R_2= R_3= R_4= R_5= 0$, and $\delta(k)=0$ for $1\leq k\leq 5$). Then, $s_1=3*2^5= 96$, $s_2= 13+ 13+ 7=33$ and $s_3= 2$, thereby resulting in $N^{(1)}(SG)=65$.

To determine $N^{(2)}(SG)$, we first determine $M^{(2)}(B) \; \forall \; B \subseteq SG$. To get $M^{(2)}(\{G^1, G^2\})$, we have $W = \{1, 2\}$, $g_{y_1}^1=2$, $g_{y_2}^2=1$, $c^1=\{3, 4, 5\}$ and $c^2=\{3, 4\}$, leading to $M^{(2)}(\{G^1, G^2\})=2^2$. We also get $M^{(2)}(\{G^1, G^3\})=2^2$ ($W=\{1, 3\}$), $M^{(2)}(\{G^2, G^3\})=0$ ($W=\{2, 3\}$), and $M^{(2)}(SG)=0$ ($W=\{1, 2, 3\}$ and $|W| > 2$). $N^{(2)}(SG)$ is thus obtained from $N^{(2)}(SG) = 3*2^4 - (4+4) = 40$. Following the same procedure, we get $M^{(3)}(\{G^1, G^2\})=2$, $M^{(3)}(\{G^1, G^3\})=2$ and $M^{(3)}(\{G^2, G^3\})=0$, resulting in $N^{(3)}(SG) = 3*2^3 - (2+2) = 20$, and $M^{(4)}(\{G^1, G^2\})=2$, $M^{(4)}(\{G^1, G^3\})=2$ and $M^{(4)}(\{G^2, G^3\})=0$, leading to $N^{(4)}(SG) = 3*2^2 - (2+2) = 8$.

Using the above procedure, one can determine the percentage of recognizable subcubes of each dimension by a set of GC's, i.e., $N^{(k)}(SG)$ out of $C_k^n 2^{n-k}$. It can be verified by Figure

6.9 that 65 (out of 80) Q_1's, 40 (out of 80) Q_2's, 20 (out of 40) Q_3's, and 8 (out of 10) Q_4's can be recognized by this strategy, which agrees with the results computed from their parameter sets. Note that one set of GC's may be better than another set of GC's in recognizing subcubes of one dimension while being worse in recognizing subcubes of another dimension. For example, let SG_A be the set of three GC's given in Figure 6.9, and SG_B= $\{G^1, G^2, G^3\}$ with parameters $\{g_j^1\}_{j=1}^5$ = {1, 2, 3, 4, 5}, $\{g_j^2\}_{j=1}^5$ = {2, 3, 5, 1, 4} and $\{g_j^3\}_{j=1}^5$ = {3, 4, 2, 5, 1}. Then, we have $N^{(1)}(SG_A)$=65 < $N^{(1)}(SG_B)$=66 and $N^{(2)}(SG_A)$=40 < $N^{(2)}(SG_B)$=42, but $N^{(4)}(SG_A)$=8 > $N^{(4)}(SG_B)$=6. This fact implies that not only the number of GC's used but also their identity affects the subcube recognition ability of an allocation strategy. Furthermore, it is important to see that when the number of GC's to be used is given, the determination of an optimal (in the sense that the probability of a successful allocation is maximized) set of GC's depends on the distribution of dimensions of the subcubes required by incoming requests.

6.2.3. Simulation results

Various processor allocation strategies in a Q_5 have been investigated via simulation. Specifically, the performance of the GC strategy is compared with that of the buddy strategy via simulation. Also, the performance improvement of an allocation strategy with multiple GC's over the one with a single GC is discussed.

6.2.3.1. Performance comparison of GC and buddy strategies

Both the buddy and GC strategies in a Q_5 are programmed using Fortran 77 on the NCUBE/six multicomputer. An incoming request is assumed to arrive every time unit. The dimensions or sizes of subcubes required by incoming requests are assumed to follow a given

distribution. The *subcube residence time* -- the length of time each allocated subcube is occupied -- is determined by a uniform distribution. Requests are generated and then allocated by using both strategies. When no subcubes of the required size are available to an incoming request, the request is queued until processor relinquishment makes room for it. Generation, allocation and queueing of requests will continue as described above until a predetermined time limit, T, is reached.

Performances of both strategies are measured in terms of *d*, the average delay in honoring a request. An alternative measure is *efficiency* $E = \sum_{i=1}^{r} 2^{|I_i|}\, t_i/(M*T)$, where $|I_i|$ and t_i are respectively the dimension and residence time of a subcube required by I_i, M is the total number of processor nodes in the system (2^5 in a Q_5), and r the number of requests honored within T. To avoid unbounded request queues, the inputs to our simulation are subject to the constraint: $E(2^{|I_i|})E(N) \le 2^5$, where $E(2^{|I_i|}) = \sum_{k=0}^{4} p_k 2^k$, p_k is the probability of a request asking for a Q_k, and E(N) is the expected number of incoming requests during every subcube residence time. As shown in Table 6.2, the GC strategy outperforms the buddy strategy. Notice that the subcube residence time for each simulation is determined in such a way that the values of $E(2^{|I_i|})E(N)$ for the three simulations are approximately the same. Observe that *E* and *R* can be improved by using the GC strategy, especially when incoming requests tend to ask for small subcubes. Intuitively, the system will be more fragmented as the allocation and deallocation of small subcubes are more frequent than those of large subcubes, generating an environment in which the GC strategy outperforms the buddy strategy with its better subcube recognition ability. This agrees well with Table 6.2.

Measurement† / Distribution††	E_{GC}	E_{BD}	d_{GC}	d_{BD}
Uniform distribution, [3, 7] p_0=.200, p_1=.200, p_2=.200, p_3=.200, p_4=.200	.775	.768	7.97	8.28
Normal distribution I, [3.83, 7.83] p_0=.098, p_1=.214, p_2=.376, p_3=.214, p_4=.098	.787	.773	7.16	8.05
Normal distribution II, [8.36, 12.36]††† p_0=.504, p_1=.217, p_2=.143, p_3=.087, p_4=.049	.763	.744	7.43	8.42

† A parameter with a subindex GC (BD) is measured under the GC (buddy) strategy.

†† [a, b] below means the subcube residence time is determined by a uniform distribution from a to b time units.

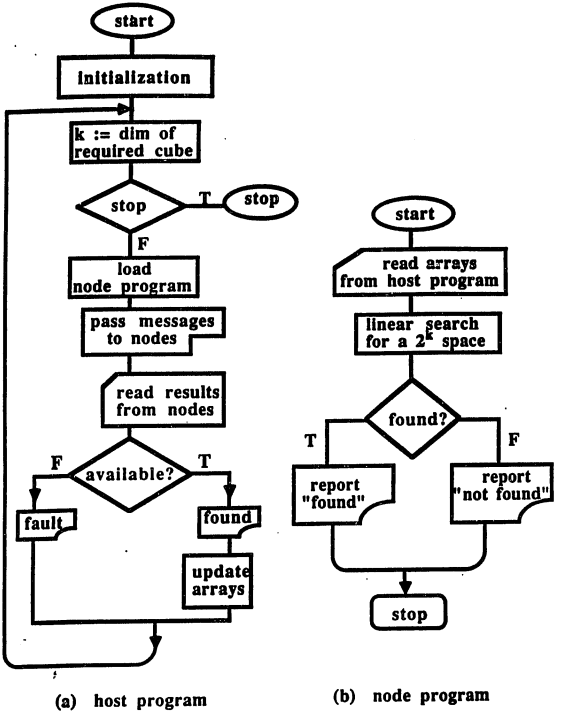
††† Only the distribution on the right half plane is considered.

Table 6.2. Simulation results of the GC and buddy strategies with T = 100.

6.2.3.2. Performance of strategies with multiple GC's

Using the procedure in Section 6.2.1, the GC's required for complete subcube recognition, as shown in Figure 6.12, are determined to be 12345, 25134, 24153, 31452, 34251, 51324, 41523, 23514, 45213 and 35412. Note that these GC's can be determined off-line and incorporated in the processor allocation algorithm. The entire program consists of the host and node subprograms. A flowchart of the entire program is given in Figure 6.13. Whenever a subcube needs to be allocated, the host program loads nodes with a search program and passes node programs the required messages, each of which is associated with a different GC and, thus, different from others. Each node program will then search for a required subcube independently of others and return the result of search to the host program once it completes the search. The request will be honored with the subcube found first by any of the participating nodes. For processor relinquishment, only the list of allocation bits kept in the host program is updated and no hypercube node is used. Note that since the number of GC's used is the same as the number of hypercube nodes performing the search, the operational overhead of an allocation strategy with multiple GC's is linearly proportional to the number of GC's used within a subcube of a fixed dimension. (This is also true for an alternative approach that all the GC's used are searched sequentially by the host.)

Insofar as the operational overhead is concerned, the number of GC's used must be reduced even if it degrades subcube recognition ability. Thus, our allocation strategy is implemented to facilitate the flexibility in adjusting subcube recognition ability, and so designed that if k GC's are to be used, then the first k GC's will be employed. Suppose p_0=.45, p_1=.25, p_2=.15, p_3=.10, p_4=.05 and p_5=.00, and the order of the ten GC's for a Q_5 is: $\{$ {1,2,3,4,5}, {4,5,2,1,3}, {3,5,4,1,2}, {2,5,1,3,4}, {3,4,2,5,1}, {5,1,3,2,4}, {4,1,5,2,3}, {2,4,1,5,3}, {3,1,4,5,2}, {2,3,5,1,4} $\}$. The subcube recognition abilities of SG_h, $1 \le h \le 10$, are shown



(a) host program (b) node program

Figure 6.13. A flowchart for the processor allocation algorithm using multiple GC's.

in Table 6.3. The percentage of recognizable subcubes of each dimension is plotted in Figure 6.14. By comparing the subcube recognition ability of the three GC's given in Figure 6.9 and that of SG_3 in Table 6.3, one can see the importance of choosing an optimal set of GC's.

It is also worth mentioning that the following proposition is very useful in formulating the mapping of a binary string to a GC with given parameters.

Proposition 6.3: The binary string $b_n b_{n-1} \cdots b_1$ is the number $\sum_{i=1}^{n} b_{g_i}^* 2^{i-1}$ in a GC with parameters g_i, i=1,...,n, where $b_{g_i}^*$'s are determined from $b_n b_{n-1} \cdots b_1$ by the following procedure:

```
for i = n to 1 step = –1 do
begin
    b*_{g_i} := b_{g_i};
    if (b*_{g_i} = 1) and (i ≠ 1) then b_{g_{i-1}} := b̄_{g_{i-1}};
end
```

For example, 10111 is the number 10 in the GC with parameters g_1=2, g_2=5, g_3=1, g_4=3 and g_5=4, since $b_4 b_3 b_1 b_5 b_2$ = 01111 and $b_4^* b_3^*\ b_1^* b_5^* b_2^*$ = 01010.

Let p_0=.45, p_1=.25, p_2=.15, p_3=.10, p_4=.05 p_5=.00, T=100, and the ten GC's be sorted in the order as stated before. Suppose the subcube residence time is determined by a uniform distribution from 4 to 12 time units. The measured performance of an allocation strategy with multiple GC's is plotted in Figure 6.15, where R is the percentage of incoming requests that are allocated successfully to required subcubes at the time of their arrival and n is the number of GC's used. A failure in allocating an available subcube to an incoming request is called a *fault*. Due to their incomplete subcube recognition ability, some of allocation strategies may generate a fault even if subcubes of the size required by an incoming request are available. Faults of this kind are termed *pseudo faults*. As shown in Figure 6.16, one can see that the number of occurrence of pseudo faults, denoted by f, decreases as the number of GC's

$N^{(k)}(SG_h)$ \ SG_h	h=1	h=2	h=3	h=4	h=5	h=6	h=7	h=8	h=9	h=10
k=0	32	32	32	32	32	32	32	32	32	32
k=1	32	57	72	79	79	80	80	80	80	80
k=2	16	32	47	58	65	71	75	78	79	80
k=3	8	16	22	25	31	34	37	38	39	40
k=4	4	8	10	10	10	10	10	10	10	10
k=5	1	1	1	1	1	1	1	1	1	1
$H(SG_h)$ ‡	.620	.768	.868	.918	.946	.968	.983	.991	.995	1.000

‡ $H(SG_h) = \sum_{k=0}^{n} \frac{N^{(k)}(SG_h)p_k}{C_k^n 2^{n-k}}$, where p_0=.45, p_1=.25, p_2=.15, p_3=.10, p_4=.05, p_5=.00 and n=5.

Table 6.3. The number of subcubes recognizable by SG_h, $1 \le h \le 10$.

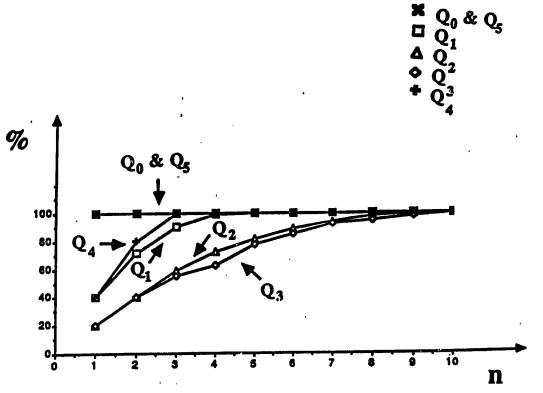
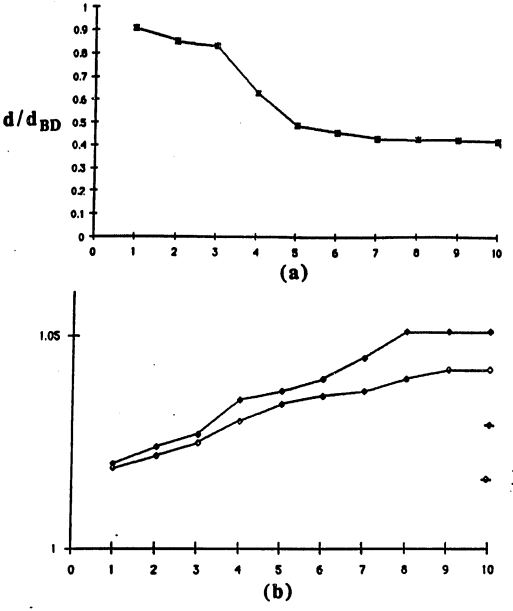


Figure 6.14. The percentage of recognizable subcubes of each dimension by an allocation strategy using multiple GC's.



d_{BD} , R_{BD} and E_{BD} are parameters measured under the buddy strategy.

Figure 6.15. Performance of an allocation strategies using multiple GC's.

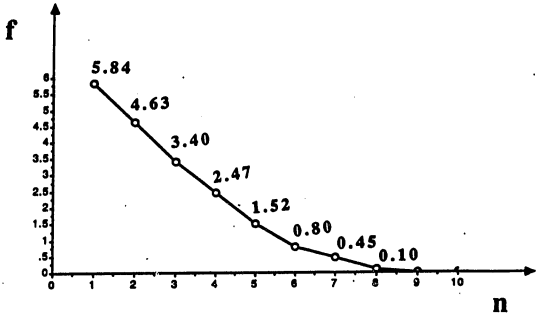


Figure 6.16. The number of pseudo faults in an allocation strategy using multiple GC's.

increases.

The performance of an allocation strategy with multiple GC's does not improve linearly with the associated search overhead. Note that a contiguous region in one GC could be several separate small regions in another GC. This fact in turn implies that the existence of a region resulting from a strategy with a single GC is more likely to be detected by the search guided by this GC than by any other GC. Thus, the improvement achieved by using multiple GC's is limited. Nevertheless, as a design problem, an allocation strategy with multiple GC's, although its necessity requires further justification in the case when the search overhead is a major concern, possesses its importance when the system utilization is important.

6.3. Task Migration under the GC Strategy

In the following three subsections we shall determine, respectively, the goal configuration, the node-mapping between the source and destination subcubes, and shortest deadlock-free paths for task migration.

6.3.1. Determination of goal configuration

There are many ways conceivable to relocate active tasks and compact occupied subcubes. However, since it is desirable to perform the task migration between each pair of subcubes in parallel, it is very important to avoid any deadlock during the migration. Clearly, a deadlock might occur if there is a circular wait among nodes. To prevent this, a linear ordering of hypercube nodes is established in such a way that each node can only move its task module to a node with a lower address, i.e., a node with address $G_n(p)$ sends its task module to another node with address $G_n(q)$, only if $p > q$. Thus, given a configuration of occupied subcubes, the goal configuration without fragmentation can be determined by the algorithm

below.

Algorithm A_1: Determination of the goal configuration

Step 1. Label each task in the availability list with a distinct number in such a way that each task allocated to a subcube with a lower address is labeled with a smaller number.

Step 2. Relocate all tasks according to an increasing order of their labels.

For example, the goal configuration in Figure 6.17b can be derived from the initial fragmented configuration in Figure 6.17a. It can be easily verified that using algorithm A_1 each task will always be moved from a subcube with a higher address to a subcube with a lower address, while a task with a smaller label is not necessarily ahead of a task with a larger label in the goal configuration. As it was proved in Theorem 6.1, the GC strategy is statically optimal. Recall that an allocation strategy is said to be statically optimal if using the strategy a Q_n can accommodate any input request sequence $\{I_i\}_{i=1}^{k}$ iff $\sum_{i=1}^{k} 2^{|I_i|} \leq 2^n$, where $|I_i|$ is the subcube dimension required by request I_i. This fact implies that fragmentation will definitely be removed by A_1, i.e., the resulting configuration can accommodate a Q_k if the number of available nodes in a Q_n is greater than or equal to 2^k.

6.3.2. Node-mapping between source and destination subcubes

Once the goal configuration is determined, each active task will be moved from its current or source subcube to the destination subcube. As mentioned earlier, the action for a node to move its task module to one of its neighboring nodes is called a moving step. To determine the number of moving steps for a task migration, we first define the *moving distance* of a task between two subcube locations as follows.

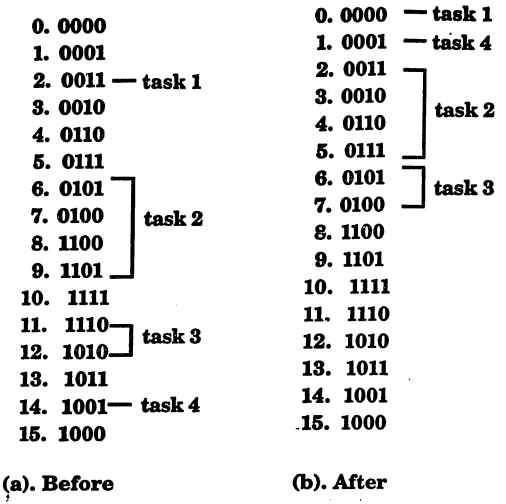


Figure 6.17. Task migration under the GC strategy.

Definition 6.2: The moving distance of a task between two subcube locations with addresses $\alpha = a_n a_{n-1} \cdots a_1$ and $\beta = b_n b_{n-1} \cdots b_1$ in a Q_n, $M : \sum^n \times \sum^n \rightarrow I^+$, is defined as

$$M(\alpha, \beta) = \sum_{i=1}^{n} m(a_i, b_i), \text{ where } m(a_i, b_i) = \begin{cases} 1, \text{ if } a_i = \overline{b_i} \neq *, \\ 0, \text{ if } a_i = b_i, \\ \frac{1}{2}, \text{ otherwise.} \end{cases}$$

Then, we have the following theorem for the minimal number of moving steps required to move a task from one subcube location to another.

Theorem 6.5 : Let $T(\alpha, \beta)$ be the minimal number of moving steps from a subcube location α to another location β. Then, $T(\alpha, \beta) = M(\alpha, \beta)2^{|\alpha|}$, where $|\alpha| = |\beta|$ is the dimension of the subcube.

To facilitate the proof of Theorem 6.5, it is necessary to introduce the following proposition whose proof is omitted.

Proposition 6.4 : Given a node $u \in Q_n$, $\sum_{w \in Q_n} H(u,w) = n2^{n-1}$.

Proof of Theorem 6.5: We shall prove $T(\alpha, \beta) \geq M(\alpha, \beta)2^{|\alpha|}$ first. Suppose $\alpha = a_n a_{n-1} \cdots a_1$ and $\beta = b_n b_{n-1} \cdots b_1$. We define the *frontier subcube* of α towards β, denoted by $\sigma_{\alpha \rightarrow \beta}(\alpha) = f_n f_{n-1} \cdots f_1$, in such a way that, $\forall$ i, $f_i = b_i$ if $a_i = *$ and $b_i \in \{0, 1\}$, and $f_i = a_i$ otherwise. For example, if $\alpha = 00{**}$ and $\beta = 1{*}1{*}$, then $\sigma_{\alpha \rightarrow \beta}(\alpha) = 001{*}$ and $\sigma_{\beta \rightarrow \alpha}(\beta) = 101{*}$. Clearly, $\sigma_{\alpha \rightarrow \beta}(\alpha)$ contains all the nodes in α which are closest to β. In addition, we define the Hamming distance between two subcubes as the shortest distance between any two nodes which respectively belong to the two subcubes, i.e., $H^*(\alpha, \beta) = \min_{u \in \alpha,\, w \in \beta} H(u, w)$. Since we align some bits of α with their corresponding bits of β to obtain $\sigma_{\alpha \rightarrow \beta}(\alpha)$, it is easy to see that $\forall$ $u \in \alpha$ and $w \in \beta$, $H(u, w) \geq H^*(u, \sigma_{\alpha \rightarrow \beta}(\alpha)) +$

$H^*(\sigma_{\alpha\to\beta}(\alpha), \sigma_{\beta\to\alpha}(\beta)) + H^*(\sigma_{\beta\to\alpha}(\beta), w)$.

Let $u' \in \beta$ denote the node to which the task module originally located at $u \in \alpha$ is to be moved. Notice that the number of moving steps required to move a task module from u to u′ is greater than or equal to the Hamming distance between them, H(u, u′). Then, $T(\alpha, \beta) \geq \sum_{u\in\alpha} H(u, u')$. Moreover, from the above reasoning, we get:

$$\begin{aligned} T(\alpha, \beta) &\geq \sum_{u\in\alpha} H(u, u') \\ &\geq \sum_{u\in\alpha} \{H^*(u, \sigma_{\alpha\to\beta}(\alpha)) + H^*(\sigma_{\alpha\to\beta}(\alpha), \sigma_{\beta\to\alpha}(\beta)) + H^*(\sigma_{\beta\to\alpha}(\beta), u')\} \\ &= \sum_{u\in\alpha} H^*(u, \sigma_{\alpha\to\beta}(\alpha)) + 2^{|\alpha|}H^*(\sigma_{\alpha\to\beta}(\alpha), \sigma_{\beta\to\alpha}(\beta)) + \sum_{u\in\alpha} H^*(\sigma_{\alpha\to\beta}(\alpha), u'). \end{aligned}$$

Let r_1 be the number of dimensions in which $\{a_i,b_i\} = \{0,1\}$, r_2 the number of dimensions in which $a_i = b_i = *$, r_3 the number of dimensions in which $a_i = *$ and $b_i \in \{0,1\}$, and r_4 the number of dimensions in which $b_i = *$ and $a_i \in \{0,1\}$. Clearly, $r_1 = H^*(\sigma_{\alpha\to\beta}(\alpha), \sigma_{\beta\to\alpha}(\beta))$, $r_2 + r_3 = |\alpha|$, $r_1 + r_3 = M(\alpha, \beta)$, and $r_3 = r_4$, because the addresses of α and β have the same number of *'s.

Therefore,

$$\begin{aligned} T(\alpha, \beta) &\geq \sum_{u\in\alpha} H^*(u, \sigma_{\alpha\to\beta}(\alpha)) + 2^{|\alpha|}H^*(\sigma_{\alpha\to\beta}(\alpha), \sigma_{\beta\to\alpha}(\beta)) + \sum_{u\in\alpha} H^*(\sigma_{\alpha\to\beta}(\alpha), u') \\ &= 2^{|\alpha|}H^*(\sigma_{\alpha\to\beta}(\alpha), \sigma_{\beta\to\alpha}(\beta)) + 2\sum_{u\in\alpha} H^*(u, \sigma_{\alpha\to\beta}(\alpha)) \\ &= 2^{|\alpha|}r_1 + 2(2^{r_2}2^{r_3-1}r_3) \qquad \text{(From Proposition 6.4 and } r_2 = |\sigma_{\alpha\to\beta}(\alpha)|.) \\ &= 2^{|\alpha|}(r_1 + r_3) = M(\alpha, \beta)2^{|\alpha|}. \end{aligned}$$

Next we prove the inequality $T(\alpha, \beta) \leq M(\alpha, \beta)\, 2^{|\alpha|}$ by showing the existence of a one-to-one mapping between nodes in α and β, and the Hamming distance between each pair

of mapping and mapped nodes is $M(\alpha, \beta)$. Suppose $p_1, p_2, \cdots, p_{r_3}$ are those dimensions in which $a_{p_i} \in \{0,1\}$ and $b_{p_i} = *$, and $q_1, q_2, \cdots, q_{r_4}$ are those dimensions in which $a_{q_i} = *$ and $b_{q_i} \in \{0,1\}$. Note that $r_3 = r_4$. Each node $u = u_n u_{n-1} \cdots u_1 \in \alpha$ can then be mapped to a node $w = w_n w_{n-1} \cdots w_1 \in \beta$ in such a way that when $i \neq p_j$ for $1 \leq j \leq r_3$,

$$w_i = \begin{cases} b_i, \text{ if } b_i \in \{0, 1\}, \\ u_i, \text{ if } a_i = b_i = *, \end{cases} \quad \text{and}$$

$$w_{p_j} = \begin{cases} \overline{u_{p_j}}, \text{ if } w_{q_j} = u_{q_j}, \\ u_{p_j}, \text{ if } w_{q_j} \neq u_{q_j}, \end{cases} \quad \text{for } 1 \leq j \leq r_3.$$

This is a one-to-one mapping, since the possibility of a many-to-one mapping is eliminated by different assignments of bits in the p_j-th dimension, $1 \leq j \leq r_3$. Moreover, we have $H(u,w) = r_1 + r_3 = M(\alpha, \beta)$. By the above node-mapping, we can determine, for each source node in α, the corresponding mapped node in β, and the total number of moving steps is $M(\alpha, \beta)\, 2^{|\alpha|}$, thus satisfying to $T(\alpha, \beta) \leq M(\alpha, \beta) 2^{|\alpha|}$. **Q.E.D.**

The above theorem proves that the minimal number of moving steps required to move a task from another location α to a subcube location β is $M(\alpha, \beta)2^{|\alpha|}$. There may be many ways to move a task from one subcube to another, each having the same total number of moving steps. For example, we can move an active task from 10*1 to 000* by either (a). $1011 \rightarrow 0000$ (3 hops) and $1001 \rightarrow 0001$ (1 hop), or (b). $1011 \rightarrow 0001$ (2 hops) and $1001 \rightarrow 0000$ (2 hops). The total number of moving steps in either case is 4. However, in order to exploit the inherent parallelism, we naturally want the total $M(\alpha, \beta)2^{|\alpha|}$ moving steps to be equally distributed among all pairs of source and destination nodes such that every node u in α requires exactly $M(\alpha, \beta)$ moving steps to transfer its task module to the corresponding node in β. Clearly, this can be accomplished by the node-mapping scheme introduced in the proof

of Theorem 6.5. Notice that the source and destination subcubes for the task migration under a strategy must belong to those subcubes recognizable by that strategy. In light of this fact, a simplified node-mapping scheme will be introduced in Corollary 6.5.1. However, it is necessary to introduce the following proposition first, which follows directly from Theorem 6.2.

Proposition 6.5: Suppose $\alpha = a_n a_{n-1} \cdots a_1$ and $\beta = b_n b_{n-1} \cdots b_1$ are two k-dimensional subcubes recognizable by the GC strategy. Then, there is at most one dimension, say p, in which $a_p \in \{0, 1\}$ and $b_p = *$.

The node-mapping between two subcubes recognizable by the GC strategy can be determined as follows. Suppose $\alpha = a_n a_{n-1} \cdots a_1$ is the source subcube and $\beta = b_n b_{n-1} \cdots b_1$ the destination subcube. Let p and q be the dimensions in which $a_p \in \{0,1\}$ and $b_p = *$, and $a_q = *$ and $b_q \in \{0,1\}$.

Corollary 6.5.1: Each source node $u = u_n u_{n-1} \cdots u_1 \in \alpha$ can be one-to-one mapped to a destination node $w = w_n w_{n-1} \cdots w_1 \in \beta$ in such a way that when $i \neq p$,

$$w_i = \begin{cases} b_i, \text{ if } b_i \in \{0, 1\}, \\ u_i, \text{ if } a_i = b_i = *, \end{cases} \quad \text{and} \quad w_p = \begin{cases} \overline{u_p}, \text{ if } w_q = u_q, \\ \text{if } w_q \neq u_q, \end{cases}$$

and $H(u,w) = M(\alpha, \beta)$.

For example, when $\alpha = 1*1*$, $\beta = 00**$ and $u = 1110$, we have $p = 3$ and then $w = 0010$. It can be verified that every node in 1*1* will need exactly 2 moving steps to relocate its task module to the corresponding node in 00**, i.e., 1010 (12) $\rightarrow$ 0000 (0), 1011 (13) $\rightarrow$ 0001 (1), 1110 (11) $\rightarrow$ 0010 (3) and 1111 (10) $\rightarrow$ 0011 (2). It is worth mentioning that the order of source nodes in a BRGC is not necessarily the same as that of their corresponding destination nodes after the node-mapping.

6.3.3. Determination of shortest deadlock-free routing

After the determination of node-mapping, we now want to develop the routing method to move each task module from its source node to the destination node. As mentioned earlier, in order to avoid deadlocks, a linear ordering among hypercube nodes is enforced such that each node can only move its task module to a node with a lower address. More formally, we need the following definition.

Definition 6.3: A path is said to be *shortest deadlock-free* (SDF) with respect to a coding scheme if it is a shortest path from the source node to the destination node and the reverse order of nodes in that coding scheme is preserved in the node sequence of that path.

In other words, if $C_n(i)$ and $C_n(j)$ are two nodes in a SDF path, then $C_n(i)$ is ahead of $C_n(j)$ in the path iff $i > j$. For example, the path [111, 011, 001] is a SDF path in G_3 of Figure 6.2a, whereas [111, 101, 001] is not. A coding scheme, C_n, is said to be *SDF path preserving* if $\forall\ p < q$ there exists a SDF path from $C_n(q)$ to $C_n(p)$, i.e., there exists $[C_n(q), C_n(r_1), C_n(r_2), \cdots, C_n(r_{d-1}), C_n(p)]$, such that $q > r_1 > \cdots > r_{d-1} > p$.

Once the node-mapping between each pair of source and destination subcubes is determined, each source node appends to its task module the address of destination node. Each node can then determine the next hop to route a task module by the algorithm below.

Algorithm A_2: Determination of a SDF path

Step 1. Each node compares the destination address $d=d_nd_{n-1}\cdots d_1$ with its own address $s=s_ns_{n-1}\cdots s_1$ from left to right. Let the j-th and k-th dimensions be respectively the first and second dimensions in which they differ, i.e., $s_i = d_i$ for $j+1 \le i \le n$ and $k+1 \le i \le j-1$, and $s_j \neq d_j$, $s_k \neq d_k$.

Step 2. **If** $\sum_{i=k}^{j-1} s_i$ is even **then** send the task module to a neighboring node along the k-th dimension, **else** send the task module to a neighboring node along the j-th dimension.

For example, suppose the source node is $G_4(12)=1010$ and the destination node d is $G_4(1)=0001$, then j=4 and k=2. The next node determined by A_2 is $G_4(3)=0010$ since $\sum_{i=2}^{3} s_i$ is odd, and thus, the 4-th dimension of 1010 is changed. Then, the next hop determined by the intermediate node $G_4(3)=0010$ is $G_4(2)=0011$, since we get j=2 and k=1 for s=0010 and d=0001. It can be verified that [1010 (12), 0010 (3), 0011 (2), 0001 (1)] is a SDF path. Actually, this is not a coincidence. As it will be proved later, the paths determined by A_2 must be SDF. To facilitate the proof, it is necessary to introduce the following lemma which compares the order of two BRGC numbers.

Lemma 6.2: Let $G_n(p) = a_n a_{n-1} \cdots a_1$ and $G_n(q) = b_n b_{n-1} \cdots b_1$ be two BRGC numbers. Suppose the i-th dimension is the first dimension in which $G_n(p)$ and $G_n(q)$ differ, when they are compared from left to right, i.e., $a_j=b_j$ for $n \geq j > i$ and $a_i \neq b_i$. Without loss of generality, we can assume $a_i=1$ and $b_i=0$. Then, $p > q$ iff $\sum_{j=i+1}^{n} a_j$ is even.

Proof: Consider the procedure of generating a BRGC described by the complete binary tree in Figure 6.6. As the number of bits in a BRGC increases, the corresponding tree grows. The address of every external node (leaf) is determined by the coded bits in the path from the root to the external node, and a BRGC is then obtained by the addresses of external nodes from left to right. It can be verified that every node which is reached from the root via even number of links labeled with 1 has a 0 left-child and a 1 right-child. Note that an external node further to the right is associated with a larger number in a BRGC. Thus, it is proved that

$p > q$ if $\sum_{j=i+1}^{n} a_j$ is even, and the fact that $p < q$ if $\sum_{j=i+1}^{n} a_j$ is odd follows similarly. **Q.E.D.**

For example, in the 3-bit BRGC of Figure 6.2a, $G_3(3) = 010$ appears after $G_3(1) = 001$, since the number of 1's in the left of their first different bit position in $G_3(3)$ is zero which is even, whereas $G_3(4)=110$ appears before $G_3(6)=101$ since the number of 1's in the left of their first different bit position in $G_3(4)$ is one. In light of Lemma 6.2, the following important theorem can be derived.

Theorem 6.6: The path determined by A_2 is SDF.

Proof: Let f(m) denote the number mapped into the binary string m in a BRGC, i.e., $p = f(m)$ iff $m = G_n(p)$. Since $f(s) > f(d)$, from Lemma 6.2 we know that $\sum_{i=j}^{n} s_i$ is odd. Let $u = u_n \cdots u_1$ denote the address of the next hop determined by A_2. Consider the case when $\sum_{i=k}^{j-1} s_i$ is even. Clearly, from Step 2 of A_2, $H(u,s)=1$ and $H(u,d) = H(s,d) - 1$, since $u_i=s_i$ if $i \neq k$, and $u_k = \overline{s_k} = d_k$. Also, we have $f(u) > f(d)$ since $\sum_{i=j}^{n} u_i = \sum_{i=j}^{n} s_i$ is odd, and $f(s) > f(u)$ since $\sum_{i=j}^{n} s_i + \sum_{i=k}^{j-1} s_i$ is odd.

On the other hand, in the case when $\sum_{i=k}^{j-1} s_i$ is odd, $H(u,s)=1$ and $H(u,d) = H(s,d) - 1$, since $u_i = s_i$ if $i \neq j$, and $u_j = \overline{s_j} = d_j$. Also, $f(s) > f(u)$ since $\sum_{i=j}^{n} s_i$ is odd, and $f(u) > f(d)$ since $\sum_{i=k}^{n} u_i = \sum_{i=k}^{j-1} u_i + \sum_{i=j}^{n} u_i = \sum_{i=k}^{j-1} s_i + \overline{s_j} + \sum_{i=j+1}^{n} s_i$ is odd. Therefore, in both cases, $H(u,d) = H(s,d) - 1$, $f(s) > f(u)$ and $f(u) > f(d)$. Since the above results hold for every intermediate node, this theorem follows. **Q.E.D.**

The above theorem shows that the task migration under the GC strategy can be accomplished via SDF paths. Also, the following corollary results from Theorem 6.6.

Corollary 6.6.1: The BRGC is SDF path preserving.

Note that the standard binary encoding scheme is not SDF path preserving, neither is the coding scheme given in Figure 6.3. For example, no SDF path exists from 010 to 001 in the standard binary encoding scheme, and nor does from $C_4(9) = 1000$ to $C_4(1) = 0001$ in Figure 6.3. This fact demonstrates another advantage of the GC strategy.

To illustrate the entire process of task migration, consider the fragmented configuration in Figure 6.17a. From A_1, we obtain the goal configuration in Figure 6.17b. By the node-mapping scheme developed in Section 6.3.2, we have 0011 → 0000 for task 1, 0101 → 0011, 0100 → 0010, 1100 → 0110 and 1101 → 0111 for task 2 (*10* → 0*1*), 1110 → 0101 and 1010 → 0100 for task 3 (1*10 → 010*), and 1001 → 0001 for task 4. The SDF routing can then be determined by A_2 as follows:

Task 1: 0011 → 0001 → 0000;

Task 2: 0101 → 0111 → 0011, 0100 → 0110 → 0010, 1100 → 0100 → 0110 and 1101 → 0101 → 0111;

Task 3: 1110 → 1100 → 0100 → 0101 and 1010 → 1110 → 1100 → 0100, and

Task 4: 1001 → 0001.

CHAPTER 7

CONCLUSIONS

7.1. Summary

In this dissertation, we have addressed several important issues on routing and task allocation in multicomputer systems. In Chapter 2 and Chapter 5, we have developed some important results applicable to general multicomputer systems. On the other hand, we have also focused on some specific multicomputer architectures, such as hexagonal meshes in Chapter 3 and hypercubes in Chapter 4 and Chapter 6, and obtained concrete results in light of their network topology.

We investigated the routing in wide-area computer networks and addressed the reduction and elimination of looping effects in Chapter 2. The amount of information required to be kept at each node in order to eliminate multi-node loops has been determined. Unlike most conventional methods in which the same routing strategy is applied indiscriminately to all nodes in the network, each node under the proposed strategy adopts its optimal routing strategy. We have not only developed the formulas to determine the minimal order of the routing strategy required for each node to eliminate looping completely, but also proposed a systematic procedure to strike a compromise between the operational overhead and network adaptability. The number of configurations to be evaluated is analyzed with a combinatorial approach.

Note that the order of the optimal routing strategy for each node can be determined off-line from a given network and incorporated into each node before the network executes certain

missions if the propagation delay is the main factor of link delay. The network is thus made to attain the maximal adaptability in case of link/node failures during such missions. However, when reducing operational overhead is essential and infrequent looping is tolerable, the use of a high order routing strategy has to be justified. This can be accomplished by the selection of an appropriate design objective function as addressed in Section 2.2.2.

We examined the routing in a multicomputer of regular topological structure in Chapter 3. A systematic method for continuously wrapping H-meshes, called the C-type wrapping, is presented. The C-type wrapping is then used to develop a simple addressing scheme, and efficient algorithms for routing and broadcasting for H-meshes. An H_3, called the HARTS (Hexagonal Architecture for Real-Time Systems), is currently being built at the Real-Time Computing Laboratory, The University of Michigan, for the purpose of investigating various low-level architectural and fault-tolerant issues for critical real-time applications.

We turned our attention to the subject of fault-tolerant routing in Chapter 4 and developed two adaptive fault-tolerant routing schemes for injured hypercube multicomputers. The first scheme is based on a depth-first search in which each node does not have to keep any network information except for the condition of its own links. The network information is added to the message as the message travels toward the destination. Performance of this scheme has been rigorously analyzed. We showed that this scheme is not only capable of routing messages successfully in an injured Q_n, but also able to choose a shortest path with a very high probability. However, due to the insufficient amount of information on faulty components, this scheme does not always guarantee the shortest path routing.

To ensure the shortest path routing, we proposed another routing scheme using the idea developed in Chapter 2, i.e., each node keeps some network information in a network delay table. We showed that, owing to the regularity of hypercube structure, the amount of infor-

mation to be kept in the network delay table of each node can be greatly reduced, and still leads to the shortest path routing.

In Chapter 5, we have derived the bounds for the number of acceptable task assignments for arbitrary G_T and G_P, a recursive formula for the case when G_T is a tree, and closed form expressions for more restricted cases. The knowledge of $N(G_T, G_P)$ can be applied not only for improving the state-space search of the task assignment problem but also for evaluating the importance of each system component. By comparing the number of acceptable assignments before and after removing a certain node/link in G_P, the importance of the node/link can be evaluated. Furthermore, we have extended the results on $N(G_T, G_P)$ to the completely general case (i.e., those assignments with dilations greater than one) in which two related tasks in G_T can be assigned to any two processors in G_P.

In Chapter 6, we have first investigated the properties of the buddy strategy, and then described an allocation strategy using the BRGC. While both strategies are proved to be statically optimal, the GC strategy is shown to outperform the buddy strategy by providing better subcube recognition ability. Furthermore, we have proved that the GC strategy is optimal in terms of the subcube recognition ability of a sequential search. We have also considered allocation strategies using multiple GC's and formulated the relationship between the GC's used and the corresponding subcube recognition ability. The minimal number of GC's required for complete subcube recognition in a Q_n is proved to be less than or equal to $C^n_{\lfloor \frac{n}{2} \rfloor}$, which is significantly less than n!, a brute-force enumeration.

Moreover, we have developed a procedure for task migration under the GC strategy to remove the system fragmentation. A goal configuration without fragmentation is determined in light of the static optimality of the GC strategy. The node-mapping between the source and destination subcubes is formulated and a routing procedure for obtaining shortest deadlock-

free paths for task migration is derived. Our results confirm the inherent superiority of the GC strategy over others.

7.2. Future Works

Some research topics related to this dissertation work warrant further investigation. First, the approach of using network delay tables is applicable to the fault-tolerant routing in various multicomputer systems. In light of network topology under consideration, the contents of the network delay tables can be characterized and minimized.

We developed in Chapter 3 a wrapping scheme for an H-mesh to obtain a homogeneous network where each node has degree six. Developing a general wrapping scheme for different topological structures with nodes of different degrees is a topic of much interest and significance. In addition, since the topology of the C-wrapped H-meshes is newly proposed, there are many interesting problems to be pursued further on the H-mesh architecture, such as fault-tolerant routing in an H_n which can exploit the topology of H-meshes, embedding of interacting task modules into an H_n and the application of the H-mesh to solve or reduce the complexity of some difficult problems. These topics are all closely related to the C-type wrapping and its associated routing and broadcasting algorithms.

In Chapter 5, the general formula of $N(G_T, G_P)$ for an arbitrary G_T could not be derived. As shown in Section 5.1.4, even in the restricted case when G_T is a cycle and G_P is a hypercube, we have to appeal to a nontrivial recursive formula. Clearly, the difficulty associated with the problem increases with the irregularity of the graphs involved. To derive a recursive expression or a tight bound for $N(G_T, G_P)$ for the case when G_T is arbitrary will be a challenging topic. In addition, we did not discuss in Chapter 6 when to execute task migration and the operational overhead associated with task migration. How to optimize the tradeoff

between the system admissibility and the operational overhead for task migration is also an important issue.

APPENDIX

APPENDIX

List of Symbols

APRS	The routing strategy previously used in ARPAnet, i.e., the one described in [60] (p. 2).
ACRS	The routing strategy currently used in ARPAnet, i.e., the one described in [61] (p. 2).
SP	Set of all paths in the network (p. 12).
$SP_{i,j}$	Set of all paths from N_i to N_j (p. 12).
A_i	Set of all nodes adjacent to N_i (p. 12).
$d(P_i)$	Summation of all link delays in a path P_i (p. 12).
$h(P_i)$	The number of links in a path P_i (p. 12).
$ave(P_i)$	The average link delay for links in a path P_i (p. 12).
$P_{i,j}$	A path with the shortest delay (i.e., an *optimal path*) in $SP_{i,j}$ (p. 12).
$P_{i,j-u,v}$	The shortest delay path in the set $SP_{i,j} - \{L_{u,v}\}$ (p. 12).
$H_k(P_i)$	The set of the first k nodes in the ordered sequence representation of a path $P_i \in SP$ (p. 12).
$SP^k_{i,j}$	Set of all paths from N_i to N_j, which are k-th order loop-free (p. 13).
$SL_{i,i}$	Set of loops starting and ending at N_i (p. 13).
$DL_{i,j}(m)$	The delay associated with $L_{i,j}$ at time m (p. 15).
$NT.dly_{\hat{N}j,d}(m)$	The delay from N_i via N_j to N_d in the network delay table of N_i under APRS during the time interval [m, m+1) (p. 15).

$OP_{s,d}(m)$ Path with the shortest delay from N_s to N_d in the network delay table of N_s under APRS during the time interval [m, m+1) (p. 15).

$DOP_{s,d}(m)$ The delay of $OP_{s,d}(m)$ (p. 15).

$NT.dly^k_{N_j,d}(m)$ The delay from N_i via N_j to N_d in the network delay table of N_i under the k-th order routing strategy during the time interval [m, m+1) (p. 20).

$RM^k_{i\leftarrow j,d}$ The routing message sent from N_j to N_i about the routing from N_j to N_d under the k-th order routing strategy (p. 20).

$OP^k_{s,d}(m)$ Path with the shortest delay from N_s to N_d in the network delay table of N_s under the k-th order routing strategy during the time interval [m, m+1) (p. 20).

$DOP^k_{s,d}(m)$ The delay of $OP^k_{s,d}(m)$ (p. 20).

m_k The number of time units required for N_s to obtain its new nonfaulty optimal path to N_d in case of a link failure under the k-th order routing strategy (p. 25).

$R_{i\leftarrow k,j}$ The required order of routing message sent from $N_k \in A_i$ to N_i to avoid all potential looping when $L_{i,j}$ became faulty (p. 34).

$R_{i\leftarrow k}$ The required order of routing message sent from $N_k \in A_i$ to N_i to avoid all potential looping (p. 34).

O_i^* The minimal order of routing strategy required for N_i to avoid all potential looping (p. 35).

O_i^k The order of the routing strategy adopted by N_i when the configuration is C_k (p. 39).

$R_c(k)$ The cost required per second for a node adopting the k-th order strategy to generate and process a routing message (p. 39).

$RC(C_k)$ The operational overhead per second induced under configuration C_k (p. 39).

$R_i(L_{i,j}; C_k)$ The expected number of time intervals required for an arbitrary node to obtain a new nonfaulty optimal path to any other node when $L_{i,j}$ became faulty (p. 40).

$RT(C_k)$ The expected number of time intervals for a path to recover from an arbitrary link failure under the configuration C_k (p. 40).

$SPL(N_i; C_k)$ Set of loops induced by the insufficient order of routing strategy of N_i in the configuration C_k (p. 40).

$SPL(C_k)$	Set of all potential loops under the configuration C_k (p. 40).
$L(P_i)$	Set of loops in the path P_i (p. 40).
$m_{u,v-i,j}(C_k)$	The number of time intervals required under the configuration C_k for N_u to obtain a new nonfaulty optimal path to N_v when $L_{i,j}$ became faulty (p. 41).
D_i	The distribution vector (D-vector) of node N_i (p. 42).
I_m	Set of integers $\{0, 1, 2, \cdots, m\}$ (p. 42).
H_n	A hexagonal mesh of size n with the C-type wrapping (p. 60).
m_x (m_y or m_z)	The number of moving steps from the source node to the destination node along the x (y or z) direction on a shortest path in a H_n (p. 68).
Q_n	A hypercube of dimension n (p. 83).
H(u,w)	The Hamming distance between two hypercube nodes u and w (p. 91).
$\oplus$	The exclusive operation between two binary strings (p. 91).
G_P	A processor graph (p. 126).
G_T	A task graph (p. 126).
V_α	The set of nodes in the graph G_α (p. 126).
E_α	The set of edges in the graph G_α (p. 126).
$N(G_T, G_P)$	The total number of desirable assignments which satisfy the adjacency condition (p. 127).
H_i (B_i)	Cliques (redundance sets) of a processor graph (p. 129).
U_m	An m-dimensional vector all entries of which are one (p. 129).
$f_A(V)$	The attaching function of the vector V associated with the adjacency matrix A (p. 132).
$\odot$	The product of vectors (p. 134).

$*_A$	The multiplication of vectors associated with adjacency matrix A (p. 134).
V^*	The sorted vector of the vector V (p. 134).
$W(V)$	The weight function of the vector V (p. 134).
$C(n_i)$	The set of children of a node n_i in a rooted tree (p. 138).
Y_i	The carrying vector of a node $n_i \in V_T$ (p. 138).
$T(n_i)$	The tree formed by the node n_i and its descendants (p. 139).
G_C	A communication graph (p. 162).
G_n	An n-bit Gray code which usually represents the binary reflected Gray code (BRGC) (p. 169).
$B_n(m)$	The binary representation of an integer m with n bits (p. 169).
$G_n(m)$	The BRGC representation of an integer m (p. 172).
I_i	The i-th incoming request (p. 173).
$\|I_i\|$	The dimension of a subcube required by the request I_i (p. 173).
g_k^i, $1 \le k \le n$,	The parameters of a Gray code G^i (p. 193).
b_j^i	The number in front of the number j in the parameters of G^i (p. 193).
a_j^i	The set, $\{g_u^i \mid 1 \le u < m-1\}$ (p. 193).
c_j^i	The set, $\{g_u^i \mid m < u \le n\}$ (p. 193).
$M(\alpha,\beta)$	The moving distance between two subcube aloocations α and β (p. 208).

REFERENCES

REFERENCES

[1] G. A. Anderson and E. D. Jensen, "Computer Interconnection Networks: Taxonomy, Characteristics and Examples," *ACM Comput. Surveys*, vol. 7, pp. 197-213, Dec. 1975.

[2] B. W. Arden and H. Lee, "Analysis of Chordal Ring Network," *IEEE Trans. on Comput.*, vol. C-30, no. 4, pp. 291-294, Apr. 1981.

[3] J. R. Armstrong and F. G. Gray, "Fault Diagnosis in a Boolean n Cube Array of Microprocessors," *IEEE Trans. on Comp.*, vol. C-30, no. 8, pp. 587-590, Aug. 1981.

[4] B. Becker and H. U. Simon, "How Robust is the n-Cube?," *Proc. 27-th Annual Symposium on Foundations of Computer Science*, pp. 283-291, Oct. 1986.

[5] L. N. Bhuyan and D. P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network," *IEEE Trans. on Comput.*, vol. C-33, no. 4, pp. 323-333, Apr. 1984.

[6] B. W. Boehm and R. L. Mobley, "Adaptive Routing Techniques for Distributed Communication Systems," *IEEE Trans. on Commun. Tech.*, vol. COM-17 , no. 3, pp. 340-349 , Jun. 1969.

[7] F. T. Boesch and R. E. Thomas, "On Graphs of Invulnerable Communication Nets," *IEEE Trans. on Circuit Theory*, vol. CT-17, no. 5, pp. 183-191, May 1970.

[8] S. H. Bokhari, "On the Mapping Problem," *IEEE Trans. on Comp.*, vol. C-30, no. 3, pp. 207-214, Mar. 1981.

[9] C. W. Brown and M. Schwartz, "Adaptive Routing in Centralized Computer Communication Networks," *Proc. IEEE Int'l Conf. Commun.*, pp. 47-12 to 47-16, June 1975.

[10] T. Cegrell, "A Routing Procedure for the TIDAS Message-Switching Network," *IEEE Trans. on Commun.*, vol. COM-23, no. 6, pp. 575-585, June 1975.

[11] T. F. Chan and Y. Saad, "Multigrid Algorithms on the Hypercube Multiprocessor," *IEEE Trans. on Comput.*, vol. C-35, no. 11, pp. 969-977, Nov. 1986.

[12] M. S. Chen and K. G. Shin, "Embedding of Interacting Task Modules into a Hypercube," *Hypercube Multiprocessors, M. T. Heath, editor*, pp. 122-129, 1987.

[13] M. S. Chen and K. G. Shin, "Processor Allocation in an N-Cube Multiprocessor Using Gray Codes," *IEEE Trans. on Comput.*, vol. C-36, no. 12, pp. 1396-1407, Dec. 1987.

[14] M. S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, Routing and Broadcasting in Hexagonal Mesh Multiprocessors," *IEEE Trans. on Comput.*, (To appear).

[15] M. S. Chen and K. G. Shin, "Message Routing in an Injured Hypercube," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, 1988 (To appear).

[16] M. S. Chen and K. G. Shin, "Subcube Allocation and Task Migration in a Hypercube Multiprocessor", (Submitted for publication.).

[17] E. Chow, H. S. Madan, J. C. Peterson, D. Grunwald, and D. Reed, "Hyperswitch Network for the Hypercube Computer," *Proc. of the 15th Annual International Symposium on Computer Architecture*, pp. 90-99, May 30 - June 2, 1988.

[18] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, vol. 13, pp. 57-69, Nov. 1980.

[19] W. W. Chu and L. M-T. Lan, "Task Allocation and Precedence Relations for Distributed Real-Time Systems," *IEEE Trans. on Comput.*, vol. C-36, no. 6, pp. 667-679, June 1987.

[20] NCUBE Corp., "NCUBE/ten: an overview", Beaverton, OR., Nov. 1985.

[21] P. J. Denning, "Parallel Computing and Its Evolution," *Commun. of the Assoc. Comp. Mach.*, vol. 29, no. 12, pp. 1163-1167, Dec. 1986.

[22] S. Dutt and J. P. Hayes, "On Allocating Subcubes in a Hypercube Multiprocessor," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, 1988 (To appear).

[23] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, vol. 15, pp. 50-56, June 1982.

[24] F. Ercal, J. Ramanujam, and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, 1988 (To appear).

[25] T. Y. Feng, "Data Manipulation Functions in Parallel Processors and Their Implementations," *IEEE Trans. on Comput.*, vol. C-23, no. 3, pp. 309-318, Mar. 1974.

[26] T. Y. Feng, "A Survey of Interconnection Networks," *IEEE Comput.*, pp. 12-27, Dec. 1981.

[27] J. P. Fillmore and S. G. Williamson, "Ranking Algorithms: The Symmetries and Colorations of the N-cube," *SIAM J. Comput.*, vol. 5, no. 2, pp. 297-304, June 1976.

[28] R. A. Finkel and M. H. Solomon, "Processor Interconnection Strategies," *IEEE Trans. on Comput.*, vol. C-29, no. 5, pp. 360-370, May 1980.

[29] V. V. Firsov, "On Isometric Embedding of A Graph into A Boolean Cube," *Cybernetics*, vol. 1, pp. 112-113, 1965.

[30] M. J. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE*, vol. 54, pp. 1901-1909, 1966.

[31] M. R. Garey and D. S. Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco : Freeman, 1979.

[32] E. N. Gilbert, "Gray Codes and Paths on The N-cube," *Bell System Tech. J.*, vol. 37, pp. 263-267, 1973.

[33] J. R. Goodman and C. H. Sequin, "Hypertree: A Multiprocessor Interconnection Topology," *IEEE Trans. on Comput.*, vol. C-30, no. 12, pp. 923-933, Dec. 1981.

[34] J. M. Gordon and Q. F. Stout, "Hypercube Message Routing in the Presence Faults," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, 1988 (To appear).

[35] R. L. Graham and H. O. Pollak, "On Embedding Graph in Squashed Cubes," in *Graph Theory and Application, Lecture Notes in Mathematics* **303**. Springer-Verlag (Proc. of a conference held at Western Michigan University.), May 10-13, 1972.

[36] S. L. Hakimi and I. T. Frisch, "An Algorithm for Construction of the Least Vulnerable Communication Network or the Graph with the Maximum Connectivity," *IEEE Trans. on Circuit Theory*, vol. CT-16, no. 5, pp. 229-230, May 1969.

[37] R. Halin, "A Theorem on N-connected Graphs," *J. Combinatorial Theory*, vol. 7, pp. 150-154, 1969.

[38] F. Harary, *Graph Theory.* Mass.: Addison-Wesley, 1969.

[39] F. Harary, "The Topological Cubical Dimension of a Graph," *Lecture Notes in the first Japan Conf. on Graph Theory and Applications*, June 3, 1986.

[40] I. Havel and J. Moravek, "B-Valuations of Graphs," *Czech. Math. J.*, vol. 22, pp. 338-351, 1972.

[41] W. D. Hillis, *The Connection Machine*. Cambridge, MA: MIT Press, 1985.

[42] C. T. Ho and S. L. Johnsson, "Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes," *Proc. Int'l Conf. on Parallel Processing*, pp. 640-648, Aug. 1986.

[43] E. Horowitz and A. Zorat , "The Binary Tree as an Interconnection Network: Applications to multiprocessor systems and VLSI," *IEEE Trans. on Comput.*, vol. C-30, no. 4, pp. 247-253, Apr. 1981.

[44] K. Hwang and J. Ghosh, "Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers," *IEEE Trans. on Comput.*, vol. C-36, no. 12, pp. 1450-1466, Dec. 1987.

[45] J. M. Jaffe and F. H. Moss, "A Responsive Distributed Routing Algorithm for Computer Networks," *IEEE Trans. on Commun.*, vol. COM-30, no. 7, pp. 1758-1762, Jul. 1982.

[46] M. J. Johnson, "Updating Routing Tables after Resource Failure in a Distributed Computer Network," *Networks*, vol. 14, pp. 379-391, 1984.

[47] H. Katseff, "Incomplete Hypercube," *Hypercube Multiprocessors, M. T. Heath, editor*, pp. 258-264, 1987.

[48] C. K. Kim and D. A. Reed, "Adaptive Packet Routing in a Hypercube," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, 1988 (To appear).

[49] K. C. Knowlton, "A Fast Storage Allocator," *Commun. of the Assoc. Comp. Mach.*, vol. 8, no. 10, pp. 623-625, Oct. 1965.

[50] D. E. Knuth, *The Art of Computer Programming, vol. 3*. Mass.: Addison-Wesley, 1973.

[51] Z. Kohavi, *Switching and Finite Automata Theory*. McGraw Hill, 1978.

[52] J. G. Kuhl and S. M. Reddy, "Distributed Fault Tolerance for Large Multiprocessor Systems," *Proc. 7-th Annual Int'l Symposium on Computer Architecture*, pp. 23-30, May 1980.

[53] T. C. Lee and J. P. Hayes, "Routing and Broadcasting in Faulty Hypercube Computers," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, 1988 (To appear).

[54] C. L. Liu, *Introduction to Combinatorial Mathematics*. New York: McGraw Hill, 1968.

[55] R.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Comput.*, vol. C-31, no. 1, pp. 41-47, Jan. 1982.

[56] A. W. Marshall and I. Olkin, *Inequalities: Theory of Majorization and Its Applications*. New York: Academic Press, 1969.

[57] A. J. Martin, "The Torus: An Exercise in Constructing a Processing Surface", Proc. 2nd Caltech Conf. on VLSI, pp. 527-537, 1981.

[58] J. M. McQuillan, "Adaptive Routing Algorithms for Distributed Computer Networks," *BBN Report*, no. 2831, May 1974.

[59] J. M. McQuillan, "Routing Algorithms for Computer Networks--- A Survey," *Proc. 1977 Nat'l Telecommun. Conf.*, p. 28, Dec. 1977.

[60] J. M. McQuillan and D. C. Walden, "The ARPA Network Design Decisions," *Computer Networks*, vol. 1, no. 5, pp. 243-289, Aug. 1977.

[61] J. M. McQuillan, I. Richer, and E. C. Rosen, "The New Routing Algorithm for the ARPANET," *IEEE Trans. on Commun.*, vol. COM-28, no. 5, pp. 711-719, May 1980.

[62] P. M. Merlin and A. Segall, "A Failsafe Distributed Routing Protocol," *IEEE Trans. on Commun.*, vol. COM-27, no. 9, pp. 1280-1287, Sep. 1979.

[63] D. Mills, "An Overview of the Distributed Computer Network," *Proc. Nat. Comput. Conf.*, vol. 45, pp. 523-531 , 1976.

[64] W. E. Naylor, "A Loop-Free Adaptive Routing Algorithm for Packet Switched Networks", Proc. 4th Data Commun. Symp., pp. 7-9 to 7-14, Quebec, P.Q., Canada, Oct. 1975.

[65] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto Calif.: Tioga, 1980.

[66] L. Kleinrock and H. Opderbeck, "Throughput in the ARPANET-Protocols and Measurement," *IEEE Trans. on Commun.*, vol. COM-25, no. 1, pp. 367-376, Jan. 1977.

[67] P. W. Purdom, Jr. and S. M. Stigler, "Statistical Properties of the Buddy System," *J. of the Assoc. Comp. Mach.*, vol. 17, no. 4, pp. 683-697, Oct. 1970.

[68] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of Tasks in a Distributed Processing System with Limited Memory," *IEEE Trans. on Comput.*, vol. C-28, no. 4, pp. 291-299, Apr. 1979.

[69] R. C. Read and D. G. Corneil, "The Graph Isomorphism Disease," *J. Graph Theory*, pp. 339-363 , 1977.

[70] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithm*. Englewood Cliffs, Mass.: Prentice Hall, 1977.

[71] K. A. Ross and R. B. Wright, *Discrete Mathematics*. Englewood Cliffs, NJ.: Prentice Hall, 1985.

[72] Y. Saad and M. H. Schultz, *Data Communication in Hypercubes*. Dep. Comput. Sci., Yale Univ. Res. Rep. 428/85., 1985.

[73] Y. Saad and M. H. Schultz, *Topological Properties of Hypercubes*. Dep. Comput. Sci., Yale Univ. Res. Rep. 389/85., 1985.

[74] Y. Saad and M. H. Schultz, "Topological Properties of Hypercubes," *IEEE Trans. on Comput.*, vol. C-37, no. 7, pp. 867-872, July, 1988.

[75] M. Schwartz and T. E. Stern, "Routing Technique Used in Computer Communication Networks," *IEEE Trans. on Commun.*, vol. COM-28, no. 4, pp. 539-552, Apr. 1980.

[76] C. L. Seitz, "The Cosmic Cube," *Commun. of the Assoc. Comp. Mach.*, vol. 28, no. 1, pp. 22-33, Jan. 1985.

[77] C. C. Shen and W. H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Comput.*, vol. C-34, no. 3, pp. 197-203, Mar. 1985.

[78] K. G. Shin and M. S. Chen, "Performance Analysis of Distributed Routing Strategies Free of Ping-Pong-Type Looping," *IEEE Trans. on Comput.*, vol. C-36, no. 2, pp. 129-137, Feb. 1987.

[79] K. G. Shin and M. S. Chen, "Minimal Order Loop-Free Routing Strategy," *IEEE Trans. on Comput.*, (To appear).

[80] K. G. Shin and M. S. Chen, "On the Number of Acceptable Task Assignments in Distributed Computing Systems," *IEEE Trans. on Comput.*, (To appear).

[81] J. E. Shore, "On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies," *Commun. of the Assoc. Comp. Mach.*, vol. 18, no. 8, pp. 433-440, Aug. 1975.

[82] K. S. Stevens, S. V. Robison, and A. L. Davis, "The Post Office - Communication Support for Distributed Ensemble Architectures," *Proc. 6-th Int'l Conf. on Distributed Computing Systems*, pp. 160-166, 1986.

[83] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Eng.*, vol. SE-3, no. 1, pp. 85-93, Jan. 1977.

[84] Q. F. Stout, "Mesh-Connected Computers with Broadcasting," *IEEE Trans. on Comput.*, vol. C-32, no. 9, pp. 826-830, Sep. 1983.

[85] Q. F. Stout and B. Wager, *Intensive Hypercube Communication I: Prearranged Communication in Link-Bound Machines.* CRL-TR-9-87, EECS Dept. The Univ of Michigan, 1987.

[86] W. D. Tajibnapis, "A Correctness Proof of a Topology Information Maintenance Protocol for Distributed Computer Networks," *Commun. Assoc. Comp. Mach.*, vol. 20, pp. 477-485, 1977.

[87] J. Turner, "Point-Symmetric Graphs with a Prime Number of Points," *J. Combinatorial Theory*, vol. 3, pp. 136-145, 1967.

[88] L. Tymes, "Routing and Flow Control in TYMNET," *IEEE Trans. on Commun.*, vol. COM-29 , no. 4 , pp. 287-293 , Apr. 1981..

[89] A. Varma and C. S. Raghavendra, "Fault-Tolerant Routing of Permutations in Extra-Stage Networks," *Proc. 6-th Int'l Conf. on Distributed Computing Systems*, pp. 54-61, 1986.

[90] M. Watkins, "Connectivity of Transitive Graphs," *J. Combinatorial Theory*, vol. 8, pp. 23-29, 1970.

[91] P. Wiley, "A Parallel Architecture Comes of Age at Last," *IEEE Spectrum*, pp. 46-50, Jun. 1987.

[92] R. S. Wilkov, "Analysis and Design of Reliable Computer Networks," *IEEE Trans. on Commun.*, vol. COM-20, no. 6, pp. 660-678, June 1972.

[93] L. D. Wittie, "Communication Structures for Large Networks of Microcomputers," *IEEE Trans. on Comput.*, vol. C-30, no. 4, pp. 264-273, Apr. 1981.

[94] C. Wu and T. Y. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Trans. on Comput.*, vol. C-29, no. 8, pp. 694-702, Aug. 1980.