Journal of Parallel and Distributed Computing **58**, 26–43 (1999)
Article ID jpdc.1999.1547, available online at http://www.idealibrary.com on IDEAL®

Synchronous Load Balancing in Hypercube Multicomputers with Faulty Nodes[1]

Kyungwan Nam,* Jaewon Seo,* Sunggu Lee,*,[2] and Jong Kim†

*Department of Electrical Engineering,
†Department of Computer Science and Engineering,
Pohang University of Science and Technology (POSTECH),
San 31 Hyoja Dong, Pohang 790-784, South Korea
E-mail: slee@vision.postech.ac.kr

Received February 20, 1998; revised and accepted March 18, 1999

This paper presents a new dynamic load-balancing algorithm for hypercube multicomputers with faulty nodes. The emphasis in our method is on obtaining global load information and performing task migration using "short paths" in a synchronous manner so that a minimal amount of communication overhead is required. To accomplish this, we present an algorithm for constructing a new logical topology from a hypercube topology with faulty nodes. This new topology is used to obtain the global load information and to perform task migration. Simulation results are used to evaluate the performance of our dynamic load balancing method when compared with previous methods. © 1999 Academic Press

Key Words: dynamic load balancing, fault tolerance, hypercube, multicomputer, task migration.

1. INTRODUCTION

In a highly parallel multicomputer system, a time-critical application can be executed quickly by splitting the application into several tasks that are executed in parallel. To get the maximum performance benefit out of the parallel system, the set of tasks should be scheduled on processing nodes such that the load on each processing node is approximately equal. Given that the set of tasks are available at the beginning and the execution times of the tasks can be accurately estimated, a *static load-balancing* method can be used to balance the load at each processing node. However, in the general situation in which the execution times of tasks

[1] This paper is a revised and updated version of work presented at the *International Conference on Parallel and Distributed Systems*, Seoul, Korea, in December 1997. This work was supported in part by the Korea Science and Engineering Foundation (KOSEF) under Grant 961-0101-09-01-3.

[2] Corresponding author.

0743-7315/99 $30.00

cannot be accurately estimated and tasks may arrive at any time, a *dynamic load-balancing* method becomes necessary [7, 9].

There has been much research on dynamic load-balancing strategies for distributed computing systems. However, on parallel computing systems, the dynamic load-balancing problem takes on different characteristics. First, parallel computers typically use a regular point-to-point or dynamic interconnection network, instead of a random network configuration. Second, the load imbalance in a distributed computer is due primarily to external task arrivals, whereas the load imbalance in a parallel computer is due to the uneven and unpredictable nature of tasks. In applications such as partial differential equations solvers using adaptively generated grids [6], the execution time cannot be known in advance. In general, the dynamic load-balancing problem in a parallel computer involves the reduction of the total application execution time by re-adjusting the allocation of tasks as the application progresses [7, 8, 1]. The overhead for dynamic load balancing includes the communication time required to obtain the load information and the cost of task migration.

Two types of load-balancing strategies can be identified. A *synchronous* load-balancing strategy [7, 8] is one in which the load information collection and task migration are performed in a global synchronous manner, On the other hand, an *asynchronous* load-balancing strategy [9, 5, 10] is one in which each node executes load-balancing operations independently of the other nodes. *Receiver initiated diffusion* (*RID*) and *sender initiated diffusion* (*SID*) are two examples of asynchronous load-balancing strategies. In SID, an overloaded node uses information obtained from its neighboring nodes to send extra tasks to underloaded nodes. In RID, an underloaded node uses local load information to request extra tasks from overloaded nodes.

Two examples of synchronous load-balancing strategies are the *dimension exchange method* (*DEM*) [7] and the *cube walking algorithm* (*CWA*) [8]. DEM has been proposed for the hypercube topology, but it can be extended to k-ary n-cube topologies [1]. DEM uses the load information of its neighbor nodes to balance the loads between pairs of nodes in each dimension of the hypercube. CWA is another hypercube-topology-based algorithm in which global load information is used to migrate the necessary number of tasks along each dimension. DEM shows superior performance to the asynchronous load-balancing methods. CWA requires fewer task migrations than DEM and thus has lower load-balancing overhead. Furthermore, CWA can fully balance the load. However, if the structure of the hypercube is destroyed due to the presence of faulty nodes, the CWA algorithm cannot be used because task migrations may require links which are no longer available.

In this paper, we present a new dynamic load-balancing method for hypercube multicomputers with faulty nodes. Although the basic load-balancing strategy follows the CWA [8] method, the CWA algorithm has been modified to permit the gathering of global load information and task migration, even in the presence of faulty nodes. Section 2 describes the load-balancing model assumed in this paper. Section 3 describes the previous algorithms for synchronous dynamic load-balancing. Sections 4 and 5 present the proposed load-balancing strategy. Section 6 presents simulation results, and the paper concludes with Section 7.

2. LOAD-BALANCING MODEL

The target architecture of this paper is a hypercube multicomputer with faulty nodes. When there is a load-balancing request message broadcast from a nonfaulty node, all nonfaulty nodes perform task migrations in a synchronous manner.

2.1. Basic Assumptions

It is assumed that there is a single long-running application executing on a hypercube multicomputer with faulty nodes. All nodes in the system are assumed to possess knowledge of the identity of the faulty nodes. It is assumed that node failures occurring during the execution of the target application are handled by a backup-sparing scheme such as [3], which has the effect of increasing the load at a backup processor in the case of a fault.

It is assumed that the target application can be divided into a fixed number of independent tasks and that all tasks can be executed on any processor in any sequence (a common assumption in load-balancing [2, 7, 8]). Although all tasks require approximately equal amounts of computation time, the exact execution time of each task is not known in advance due to its unpredictable nature. External task arrivals can be handled simply as changes in the load estimate for each node, which occur anyway due to load estimate updates resulting from the completion of tasks with previously uncertain execution times.

As the application progresses and tasks finish execution, an imbalance forms in the number of tasks waiting to execute at each node. When the load imbalance progresses to the point where severely underloaded nodes exist, an underloaded node initiates load-balancing by broadcasting a load-balancing request. The load balancer at each node is invoked when the load-balancing request is received. Then,

TABLE 1

Table of Mathematical Notation

Q_n	n-dimensional hypercube
Q_k	healthy k-dimensional subcube
C_b	balancing subcube
N	number of nodes in Q_n
F	set of faulty nodes
C	set of maximum size healthy subcubes
G_i	set of faulty nodes that have distance i from Q_k
$l(u_i)$	number of tasks in node u_i
$l_t(u_i)$	number of tasks in u_i and all descendant nodes of u_i
$l^j(u_i)$	number of tasks in a j-*cube* in C_b and all tree nodes attached to the j-cube
l_{avg}	average number of tasks per node
$q_t(u_i)$	task quota of u_i and all descendant nodes of u_i
$q^j(u_i)$	task quota of a j-*cube* in C_b and all tree nodes attached to the j-cube
$\mathbf{l}(u_i)$	$\{l^0(u_i), ..., l^n(u_i), l_t(u_i), l_t(child(u_i))\}$ (a load vector)
$\mathbf{q}(u_i)$	$\{q^0(u_i), ..., q^n(u_i), q_t(u_i), q_t(child(u_i))\}$ (a quota vector)
$\omega(u_i)$	unit number of descendant nodes of u_i plus 1
$child_j(u_i)$	jth child node of u_i in a tree
$parent(u_i)$	parent node of u_i in a tree

when all nodes reach an agreed-upon synchronization point, the load-balancing algorithm is executed in a synchronous manner. This type of synchronous operation facilitates the accumulation of global load information through message exchanges and also facilitates the actual transfer of overloaded tasks.

2.2. Table of Notation

Table 1 presents a table of notation for the various mathematical symbols which will be used to describe our load-balancing algorithm and previous load-balancing algorithms.

3. PREVIOUS WORK

3.1. Receiver Initiated Diffusion (RID)

RID and SID are decentralized, asynchronous load-balancing algorithms which are initiated by underloaded and overloaded nodes, respectively. It has previously been reported that the RID strategy outperforms the SID strategy [7]. Therefore, we will compare our proposed algorithm with only the RID strategy. In RID, when a processor's load drops below a prespecified threshold, the load-balancing process is activated by the underloaded node sending load-balancing requests to its neighbor nodes. The underloaded node collects information about the number of tasks owned by each of its neighbors and then calculates the number of tasks to be received from its neighbors as

$$\varphi_{i\oplus k}(i) = l_{\text{avg}} \frac{l(k)}{\sum_{k=\text{neighbor node}} l(k)}, \qquad l_{\text{avg}} = \frac{\sum_{k=\text{neighbor node}} l(k)}{\text{number of neighbor nodes} + 1},$$

where $l(i)$ is the number of tasks owned by node i, l_{avg} is the average number of tasks owned by node i and its neighbor nodes, and $\varphi_{i\oplus k}(i)$ is the number of tasks that node i wishes to receive from node k. Note that RID uses only local load information.

3.2. Dimension Exchange Method (DEM)

DEM [7] is a synchronous load-balancing algorithm proposed for the hypercube multicomputer. In DEM, load-balancing is achieved by exchanging load information along each dimension and migrating tasks between adjacent nodes based on this information. Algorithm 1 provides a formal description of the DEM algorithm.

ALGORITHM 1 (Dimension exchange method).

DEM (Q_n, l).
1. for $k = 0$ to $n-1$
2. send $l(u_i)$ to $u_{i\oplus 2^k}$
3. receive $l(u_{i\oplus 2^k})$ from $u_{i\oplus 2^k}$
4. if $(l(u_i) > l(u_{i\oplus 2^k}))$
5. send $\lfloor (l(u_i) - l(u_{i\oplus 2^k}))/2 \rfloor$ tasks to $u_{i\oplus 2^k}$
6. if $(l(u_i) < l(u_{i\oplus 2^k}))$
7. receive $\lfloor ((l(u_{i\oplus 2^k}) - l(u_i)/2 \rfloor$ tasks from $u_{i\oplus 2^k}$.

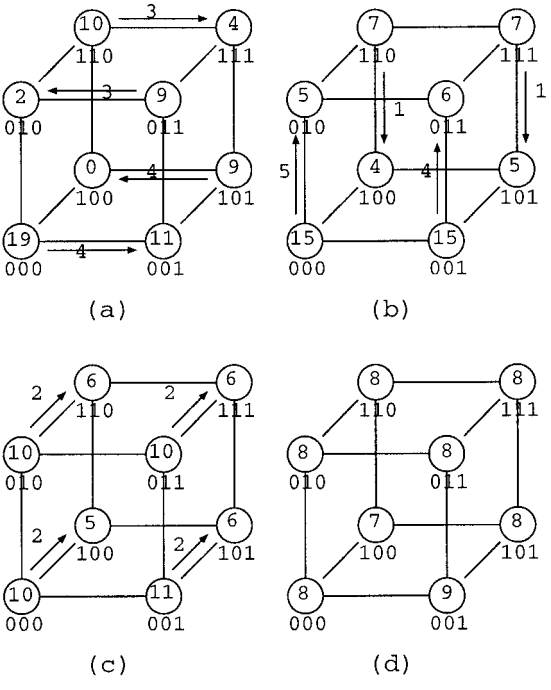

FIG. 1. Dimension exchange method [7].

Load balancing begins when an underloaded node with a load value less than a prespecified threshold broadcasts a load-balancing request. Figure 1 shows an example of the DEM algorithm in operation when load-balancing is initiated by node 100, where circled numbers represent loads and labeled arrows represent task migrations. As can be seen from Algorithm 1 and Fig. 1, $2 \log N$ steps of information exchange and $\log N$ task migrations are required for the load-balancing. However, note that the DEM algorithm does not fully balance the load—the algorithm only guarantees that the final load difference will be bounded by $\log N$, the dimension of the hypercube. Also, there are several unnecessary task migrations. If we consider dimension 0, we observe that the load imbalance between subcubes XX0 and XX1 could be resolved by migrating just one task, since subcube XX0 has 31 tasks and subcube XX1 has 33 tasks. However, due to the lack of global load information in the DEM method, 14 task migrations are performed. The CWA algorithm [8] was proposed specifically to overcome these kinds of shortcomings.

3.3. Cube Walking Algorithm (CWA)

Of the two components of load-balancing overhead (load information gathering and task migration), the task migration cost is the dominant factor as a significantly larger-sized message is required to migrate a task, as opposed to sending load information. Since the task migration cost is proportional to the total number of tasks to be migrated and the total distance to be migrated, we must attempt to minimize $\sum_k e_k$, where e_k denotes the number of tasks migrated along edge k. The minimization of this metric can be accomplished by modeling the

problem as a flow graph and solving a *minimum cost maximum flow* problem. However, this type of solution has a time complexity of $O(N^2v)$, where N is the total number of nodes and v is the required flow, which is considered to be too severe for a dynamic load-balancing algorithm [8]. Thus, the CWA method, shown as Algorithm 2 below, was proposed [8] to provide a fast heuristic solution to this problem.

ALGORITHM 2 (Cube walking algorithm).

CWA(l, Q_n)

1. for $k = 0$ to $n-1$
2. send $l^k(u_i)$ to $u_{i \oplus 2^k}$
3. receive $l^k(u_{i \oplus 2^k})$ from $u_{i \oplus 2^k}$
4. $l^{k+1}(u_i) = l^k(u_i) + l^k(u_{i \oplus 2^k})$
5. QuotaCalculation($\boldsymbol{l}(u_i)$, $\mathbf{q}(u_i)$, $\boldsymbol{\delta}(u_i)$)
6. for $k = n-1$ to 0
7. TaskMigration($\boldsymbol{l}(u_i)$, $\mathbf{q}(u_i)$, $\boldsymbol{\delta}(u_i)$, k)

QuotaCalculation($\boldsymbol{l}(u_i)$, $\mathbf{q}(u_i)$, $\boldsymbol{\delta}(u_i)$)

8. $l_{\text{avg}} = \lfloor l^n(u_i)/N \rfloor$
9. $R = l^n(u_i) \bmod N$
10. if $i < R$ $q^0(u_i) = l_{\text{avg}} + 1$
11. else $q^0(u_i) = l_{\text{avg}}$
12. for $k = 0$ to $d-1$
13. $q^k(u_i) = q^{k-1}(u_i) + q^{k-1}(u_{i \oplus 2^k})$
14. $\delta^k(u_i) = l^k(u_i) - q^k(u_i)$

TaskMigration($\boldsymbol{l}(u_i)$, $\mathbf{q}(u_i)$, $\boldsymbol{\delta}(u_i)$, k)

15. if $\delta^k(u_i) > 0$
16. $\theta^k(u_i) = \delta^k(u_i)$, $\gamma^k(u_i) = 0$
17. for $j = k-1$ to 0
18. $\theta^j(u_i) =$

$$\begin{cases} 0 & \text{if } \delta^j(u_i) \leqslant \gamma^{j+1}(u_i) \text{ and } i \wedge 2^j = 0, \\ \min(\delta^j(u_i) - \gamma^{j+1}(u_i),\ \theta^{j+1}(u_i)) & \text{if } \delta^j(u_i) > \gamma^{j+1}(u_i) \text{ and } i \wedge 2^j = 0, \\ \theta^{j+1}(u_i) & \text{if } \delta^j(u_{i \oplus 2^j}) \leqslant \gamma^{j+1}(u_i) \text{ and } i \wedge 2^j \neq 0, \\ \max(\delta^j(u_i),\ 0) & \text{if } \delta^j(u_{i \oplus 2^j}) > \gamma^{j+1}(u_i) \text{ and } i \wedge 2^j \neq 0, \end{cases}$$

where $\gamma^j(u_i) = \delta^j(u_i) - \theta^j(u_i)$

19. send $\theta^0(u_i)$ tasks to $u_{i \oplus 2^k}$ with θ vector
20. update its own vectors $l^j(u_i) = l^j(u_i) - \theta^j(u_i)$
$\delta^j(u_i) = \delta^j(u_i) - \theta^j(u_i)$
for all $j = 0, 1, ..., k$-1
21. else receive tasks and θ vector from $u_{i \oplus 2^k}$
22. update its own vectors $l^j(u_i) = l^j(u_i) + \theta^j(u_{i \oplus 2^k})$
$\delta^j(u_i) = \delta^j(u_i) + \theta^j(u_{i \oplus 2^k})$
for all $j = 0, 1, ..., k$-1

In Algorithm 2, vector $\boldsymbol{l} = \{l^n(u_i),\ l^{n-1}(u_i), ..., l^0(u_i)\}$ is a vector of the loads of all subcubes containing node u_i. If a node pair in dimension 0 adds their $l^0(u_i)$

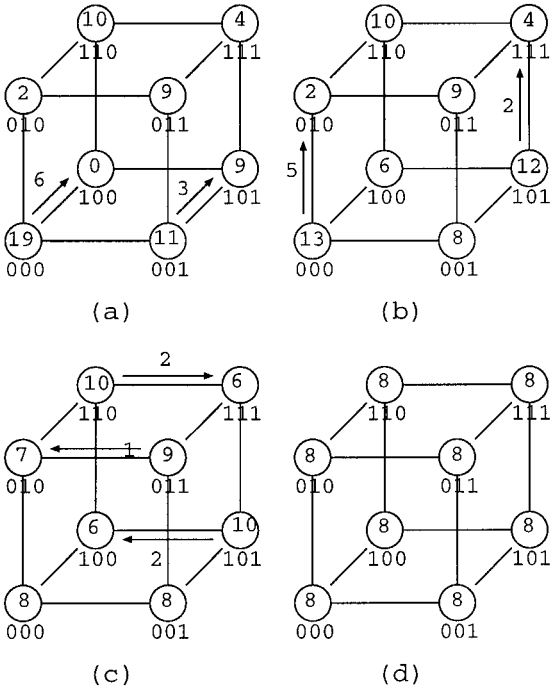

FIG. 2. Cube walking algorithm [8].

loads, the resulting load is $l^1(u_i)$, which is the load in the corresponding $1-cube$. Likewise, $l^2(u_i)$ is formed as the sum of the loads $l^1(u_i)$ of a node pair in dimension 1, and so on. "QuotaCalculation" is a function that uses the $\boldsymbol{l}$ vector to calculate the *quota* vector **q**. Then, in "TaskMigration," the difference δ between $\boldsymbol{l}$ and **q** is used to determine whether to send tasks to a neighbor or to receive tasks from a neighbor. When transferring tasks, the θ vector, which is a summary of task migration information, is sent so that global load information can be updated at the receiving node. Given an overloaded k-dimensional subcube, $\theta^k(u_i)$ tasks are divided among the member nodes in order to transfer those $\theta^k(u_i)$ tasks to an underloaded k-dimensional subcube—the load that a node u_i has to transfer is given by $\theta^0(u_i)$. This procedure is repeated for each dimension, thereby balancing the load at each node. The number of communication steps in the CWA algorithm is $3 \log N$. Example 1 shows an example of the CWA algorithm in operation, with the resulting task migrations shown in Fig. 2.

TABLE 2

Calculation Results at Nodes in Subcube 0XX

Node	l^0	δ^0	θ^0	γ^0	l^1	δ^1	θ^1	γ^1	l^2	δ^2	θ^2	γ^2
000	19	11	6	5	30	14	9	5	41	9	9	0
001	11	3	3	0	30	14	9	5	41	9	9	0
010	2	−6	0	−6	11	−5	0	−5	41	9	9	0
011	9	1	0	1	11	−5	0	−5	41	9	9	0

TABLE 3

Calculation Results for Nodes in Subcube 00X and 10X

Node	l^0	δ^0	θ^0	γ^0	l^1	δ^1	θ^1	γ^1
000	13	5	5	0	21	5	5	0
001	8	0	0	0	21	5	5	0
100	6	−2	0	−2	18	2	2	0
101	12	4	2	2	18	2	2	0

EXAMPLE 1. CWA starts when an underloaded node broadcasts a load-balancing request, in this case node 100 with 0 load. Messages are exchanged in each dimension to obtain the $\boldsymbol{l}$ vector. Then, "QuotaCalculation" is used to calculate $\mathbf{q}$ from $\boldsymbol{l}$. Then, task migration begins on the basis of the global load information obtained through the information exchanged in each dimension. The first dimension along which task migration is performed is dimension 2. Table 2 shows the θ and δ values calculated at nodes in overloaded subcube 0XX.

Using the values calculated in Table 2, tasks are migrated along dimension 2 as shown in Fig. 2(a)—the number of tasks migrated are $\theta^0(u_i) = 6$ and 3 tasks from nodes 000 and 001, respectively. Then the loads between subcubes 0XX and 1XX are balanced.

After task migration in dimension 2, the θ vector is calculated for dimension 1. Since the overloaded 1-cubes are 00X and 10X, nodes 000, 001, 100, and 101 calculate θ and δ vectors. The calculated values are shown in Table 3.

Based on the values calculated in Table 3, tasks are migrated along dimension 1 as shown in Fig. 2(b). Nodes 000 and 101 transfer θ^0 tasks in dimension 1. Next, to perform the necessary task migrations in dimension 0, the θ vector is calculated for overloaded nodes 011, 101, and 110 (shown in Table 4).

According to Table 4, in dimension 0, nodes 011, 101, and 110 should transfer 1, 2, and 2 tasks, respectively. After this final step, all nodes in the system should have approximately equal loads, as shown in Fig. 2(d).

Although CWA appears to perform well on nonfaulty hypercubes, as documented in [8], this algorithm fails to function properly if the hypercube topology is disrupted due to node failures. Problems occur in both the global load information-gathering stage and in the task migration stage. Task migration becomes a problem because some links will become effectively unusable due to the presence of the faulty nodes. The gathering of global load information becomes a problem because n stages of information exchange in each of the n dimensions of an n-dimensional

TABLE 4

Calculation Results at Nodes 011, 101, and 110

Node	l^0	δ^0	θ^0	γ^0
011	9	1	1	0
101	10	2	2	0
110	10	2	2	0

faulty hypercube will not be sufficient to gather all of the load information. Thus, a modified CWA algorithm has been devised in order to perform synchronous dynamic load-balancing on a hypercube with faulty nodes.

4. MODIFIED CUBE WALKING ALGORITHM

The *modified cube walking algorithm* (*MCWA*) works by embedding the faulty hypercube (hypercube with missing nodes) on a new logical topology and performing dynamic load-balancing on this new logical topology. Our strategy involves first finding a maximum-size healthy subcube C_b (referred to as a *balancing subcube*) within the faulty hypercube. All other nodes then attach to the balancing subcube in a tree-like manner. The resulting logical topology is shown in Fig. 3, and the method for generating this topology is described in Section 5.

Load balancing is performed using MCWA and the new logical topology in the following manner. Load balancing is initiated when an underloaded node broadcasts a load-balancing request message. Each node starts the load-balancing process in a synchronized manner, following reception of the load-balancing request message. First, all nodes must propagate their load information up to a node in the balancing subcube. Then, CWA (in which quota calculations must be weighted according to the size of the respective "trees") is used to distribute the global load information and to migrate tasks within the balancing subcube. Finally, the accumulated tasks must be migrated to the underloaded nodes in a tree. MCWA is shown below as Algorithm 3 and a detailed example (Example 2) is used to demonstrate the operation of this algorithm. Figure 4 is used to show the task migrations involved in Example 2.

ALGORITHM 3 (Modified cube walking algorithm).

MCWA($\boldsymbol{l}$, Q_n)

1. if u_i is a node in balancing subcube
2. for each child node $child_j(u_i)$
3. receive load information $l_t(child_j(u_i))$
4. $l^0(u_i) = l(u_i) + \sum_j l_t(child_j(u_i))$
5. for $k = 1$ to $\dim(C_b)$, where m is kth dimension of C_b

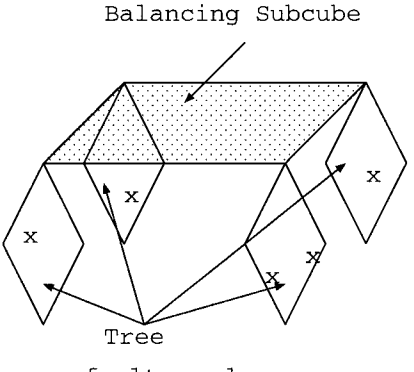

FIG. 3. New topology for dynamic load balancing.

```
6.          send l^{k-1}(u_i) to u_{i⊕2^m}
7.          receive l^{k-1}(u_{i⊕2^m}) from u_{i⊕2^m}
8.          l^k(u_i) = l^{k-1}(u_i) + l^{k-1}(u_{i⊕2^m})

9.      QuotaCalculation(l(u_i), q(u_i), δ(u_i))
10.     for each child node child_j(u_i)
11.         send quota information q_t(child_j(u_i))
12.     for each child node child_j(u_i) such that l_t(child_j(u_i)) > q_t(child_j(u_i))
13.         receive l_t(child_j(u_i)) − q_t(child_j(u_i)) tasks
14.     for k = dim(C_b) − 1 to 0

15.         TaskMigration(l(u_i), q(u_i), δ(u_i), k)
16.     for each child node child_j(u_i) such that l_t(child_j(u_i)) < q_t(child_j(u_i))
17.         send q_t(child_j(u_i)) − l_t(child_j(u_i)) tasks
18. else
19.     for each child node child_j(u_i)
20.         receive load information l_t(child_j(u_i))
21.     l_t(u_i) = l(u_i) + Σ_j l_t(child_j(u_i))
22.     send l_t(u_i) to parent(u_i)
23.     receive q_t(u_i) from parent(u_i)

24.     QuotaCalculation(l, q)
25.     for each child node child_j(u_i)
26.         send quota information q_t(child_j(u_i))
27.     for each child node child_j(u_i) such that l_t(child_j(u_i)) > q_t(child_j(u_i))
28.         receive l_t(child_j(u_i)) − q_t(child_j(u_i)) tasks
29.     if l_t(u_i) > q_t(u_i) send l_t(u_i) − q_t(u_i) tasks to parent
30.     elseif l_t(u_i) < q_t(u_i)  receive l_t(u_i) − q_t(u_i) tasks from parent
31.     for each child node child_j(u_i) such that l_t(child_j(u_i)) < q_t(child_j(u_i))
32.         send q_t(child_j(u_i)) − l_t(child_j(u_i)) tasks

QuotaCalculation(l(u_i), q(u_i), δ(u_i))
33. if u_i is a node in balancing subcube
33.     l_avg = ⌊l^{dim(C_b)}(u_i)/(number of safe node)⌋
34.     R = l^{dim(C_b)}(u_i) mod (number of safe node)
35.     if R ≠ 0 and Σ_{j=0}^{i} ω(u_j) < R, where u_j is a node in C_b
36.         q^0(u_i) = ω(u_i) l_avg + min(ω(u_i), R − Σ_{j=0}^{i} ω(u_j))
37.     else q^0(u_i) = ω(u_i) l_avg
38.     δ^0(u_i) = l^0(u_i) − q^0(u_i)
39.     for k = 1 to dim(C_b)
40.         q^k(u_i) = q^{k-1}(u_{i⊕2^k}) + q^{k-1}(u_i)
41.         δ^k(u_i) = l^k(u_i) − q^k(u_i)
42. q_t(u_i) = q^0(u_i)
43. r_t(u_i) = q_t(u_i) mod ω(u_i)
44. if r_t(u_i) ≥ 1 r_t(u_i) = r_t(u_i) − 1
45. for each child node child_j(u_i)
46.     if (r_t(u_i) − Σ_{m=0}^{j-1} ω(child_m(u_i)) > 0)
```

47. $q_t(child_j(u_i)) = \lfloor q_t(u_i)\ \omega(child_j(u_i))/\omega(u_i) \rfloor + \min(\omega(child_j(u_i)), r_t(u_i) - \sum_{m=0}^{j-1} \omega(child_m(u_i)))$
48. else $q_t(child_j(u_i)) = \lfloor q_t(u_i)\omega(child_j(u_i))/\omega(u_i) \rfloor$

MCWA operates as follows. First, in lines 2, 3, 4, 19, 20, and 21, load information is gathered within a tree by accumulating the load information of each child node. Steps 5, 6, 7 , and 8 involve forming global load information by message exchange in each dimension of C_b. Then each node calculates the quota which is a number of tasks in balanced condition. The procedure for quota calculation is shown in *QuotaCalculation*. $q_t(u_i)$ is the quota of u_i and all descendant nodes of u_i. $q^k(u_i)$ is the quota of a $k-cube$ in C_b and all tree nodes attached to that $k-cube$. Each quota value depends on the number of descendant nodes attached. In lines 10, 11, 25, and 26, the quota value calculated is sent to child nodes.

In lines 12, 13, 27, 28, and 29, nodes in trees whose loads $l_t(u_i)$ exceed their quota $q_t(u_i)$ send excess tasks to parent nodes. After the excess tasks of a tree is accumulated at a balancing subcube node, task migration in the balancing subcube starts according to CWA. Steps 14 and 15 involve task migration in the balancing subcube. Function *TaskMigration* is same as in the CWA method. In lines 16, 17, 30, 31, and 32, tasks are migrated to underloaded child nodes. Then the load in the system should be fully balanced with a difference in the number of tasks in each node of at most 1.

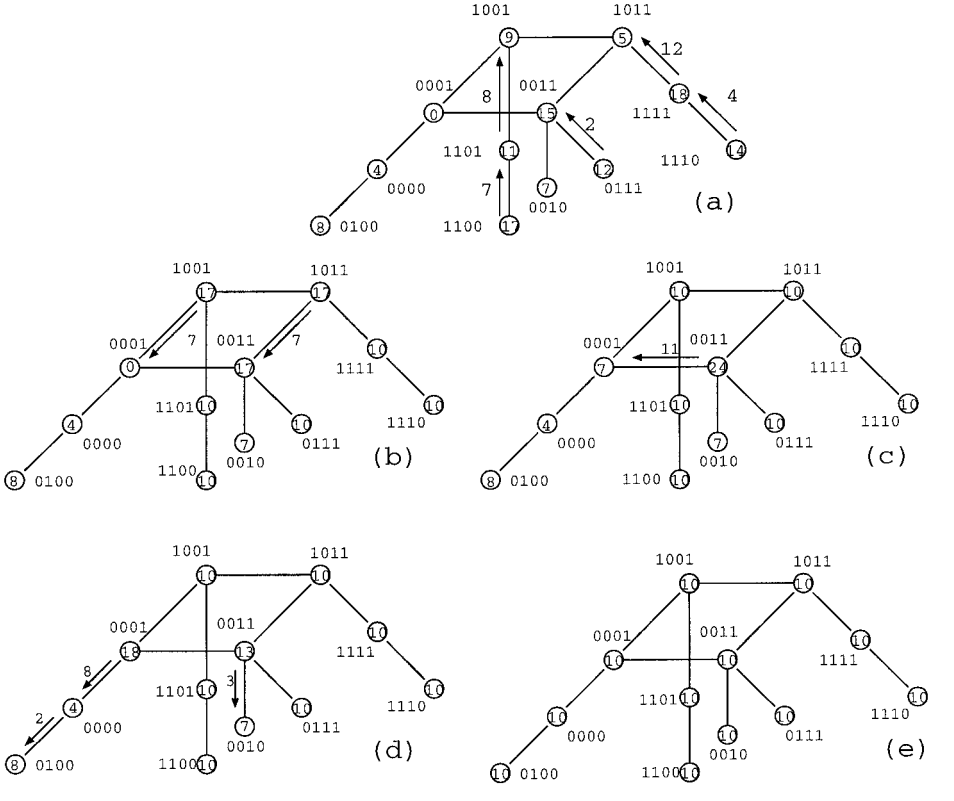

FIG. 4. Running example of modified cube walking algorithm.

TABLE 5

Calculation Results at Nodes in 10X1

Node	l^0	δ^0	θ^0	γ^0	l^1	δ^1	θ^1	γ^1
1001	37	7	7	0	74	14	14	0
1011	37	7	7	0	74	14	14	0

In this algorithm, execution of lines 2, 3, 19, 20, and 22 take T communication steps, where T is the maximum depth of all trees attached to the balancing subcube. Similarly, execution of lines (10, 11, 23, 25, and 26), lines (12, 13, and 27–29), and lines (16, 17, and 30–32) each (each set of lines in parentheses) require T steps. Execution of lines 5–7 takes $2 * (\log N - T)$ steps and execution of lines 14–15 takes $\log N - T$ steps. Therefore, the total number of communication steps is $3 \log N + T$, which is $O(\log N)$.

EXAMPLE 2. Figure 5(a) shows an example of a hypercube multicomputer with faulty nodes, 0101, 0110, 1000, and 1010. The logical topology constructed from Fig. 5(a) is shown in Fig. 4. MCWA is performed using this topology.

MCWA starts with the broadcast of a load-balancing request from node 0001. All nodes propagate load information $l_t(u_i)$ up to a node in the balancing subcube. $l_t(u_i)$ is the number of tasks in node u_i and all u_i's descendant nodes. For example, $l_t(1011)$ is 37, $l_t(1111)$ is 32, and $l_t(1110)$ is 14.

Messages are exchanged in each dimension of the balancing subcube to form a global load information vector $\boldsymbol{l}$. After global load information is acquired, a balancing subcube node calculates its quota vector, $\mathbf{q}$, and distributes its quota of child nodes. For example, $q_t(1011)$ is 30, $q_t(1111)$ is 20, and $q_t(1110)$ is 10. Nodes whose $l_t(u_i)$ exceed $q_t(u_i)$ send excess tasks to parent nodes as shown in Fig. 4(a). Then, nodes in overloaded $1-cube$ 10X1 calculate θ and send θ^0 tasks to underloaded $1-cube$ 00X1 as shown in Fig. 4(b). The calculated θ vector of $1-cube$ 10X1 is shown in Table 5.

After task migration between 10X1 and 00X1, overloaded node 0011 calculates its θ vector as shown in Table 6 and sends $\theta^0(0011)$ tasks to underloaded node 0001 as shown in Fig. 4(c).

Task migration to the underloaded tree is shown in Fig. 4(d). The load in the system is fully balanced after the execution of the MCWA algorithm, as shown in Fig. 4(e).

TABLE 6

Calculation Results at 0011

Node	l^0	δ^0	θ^0	γ^0
0011	41	11	11	0

5. GENERATION OF NEW LOGICAL TOPOLOGY

A "deep" tree in the logical topology has a negative effect on the task migration cost. The task migration cost between different tree nodes increases as the tree depth is increased. Kim [4] has proposed an algorithm which finds a healthy Q_{n-2} cube from a Q_n with faulty nodes. But in our approach, all maximum size healthy cubes are identified and the one that has the minimum tree depth is selected. The complexity of applying a breadth-first traversal algorithm to find a maximum size healthy subcube with minimum tree depth is $O(n^2 2^n)$. To reduce this complexity, we propose an algorithm, shown below as Algorithm 4, which selects a maximum size healthy subcube Q_k with tree depth $n-k$ from Q_n (with time complexity $O(n^2 N_f^2 + 2^n)$, where N_f is the number of faulty nodes). This complexity is not excessive since $2^n = N$ is simply the total number of nodes in the system.

ALGORITHM 4 (Algorithm for the generation of a new logical topology).

NewTopology(Q_n, F)
11. FindMaxHealthyCubes(F, C, k)
next: for each $Q_k \in C$
3. Group the faulty nodes into $G_1, G_2, ..., G_i, ...$, where i is the distance to a healthy subcube
4. for j such that $(1 \leqslant j \leqslant n-k)$
5. for a in G_j
6. if the number of b in G_j such that $a[i] = b[i]$ for all i satisfying $Q_k[i] =$ X is larger than j
7. go to next
8. go to find
find: BreadthFirstTraversal(Q_n, Q_k.

An example of the proposed algorithm is shown in Fig. 5. There are four maximum-size healthy cubes, XX11, X0X1, 1XX1, and 11XX. Function *FindMaxHealthyCubes* finds these healthy cubes. Faulty nodes are grouped into G_1, G_2, according to the distance from each healthy subcube. The case for XX11 is shown in Fig. 5(b). In the case of XX11, nodes 0110 and 0101 in G_1 satisfy the condition of step 6. Thus function *NewTopology* determines that XX11 is not adequate as a balancing subcube and examines the next healthy subcube, X0X1.

The reason why XX11 is not adequate as a balancing subcube is that faulty nodes 0110 and 0101 disconnect the minimum path from 0100 to XX11. The purpose of this algorithm is to exclude healthy subcubes which cannot connect to nodes using minimum length paths. This algorithm selects X0X1 as the balancing subcube as shown in Fig. 5(c). The tree depth of the new logical tree is 2.

THEOREM 1. *The tree depth of a new logical topology generated by Algorithm* 4 *is at most* $n-k$ *when the maximum-size healthy subcube is a k-cube.*

Proof. Given an arbitrary node u_i, assume that all u_i's minimum length paths to a healthy subcube Q_k selected by Algorithm 4 are disconnected by faulty nodes. If the length of a disconnected minimum path is j $(2 \leqslant j \leqslant n-k)$, there are faulty

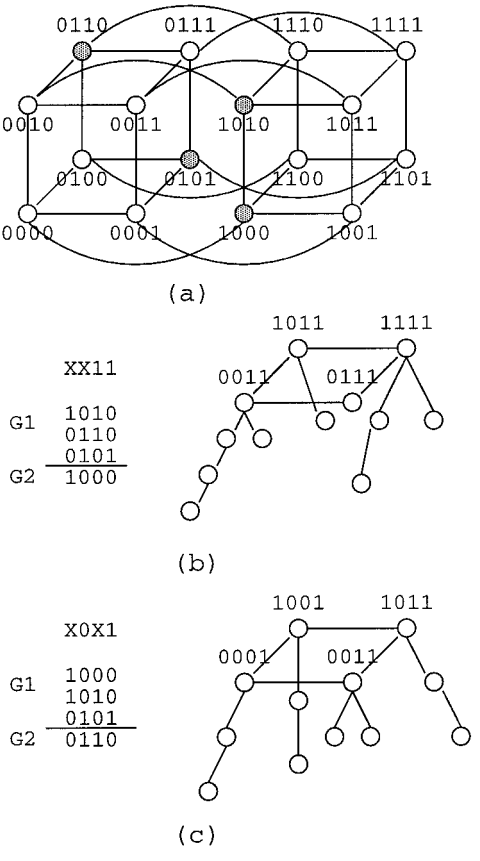

FIG. 5. Selecting balancing subcube from candidate healthy cube.

nodes whose number is not less than j adjacent to u_i, and the length of their minimum path to Q_k is $j-1$. But Algorithm 4 excludes all such Q_k. Thus all nodes are connected to Q_k with minimum paths of length at most $n-k$.

6. SIMULATION RESULTS

We tested MCWA, DEM, and RID algorithms for hypercube topologies of various sizes with various numbers of faulty nodes. CWA could not be compared with our algorithm as CWA does not work properly in the presence of faulty nodes.

All dynamic load balancing strategies begin when an underloaded node broadcasts a load-balancing request. If a neighbor node is a faulty node, DEM performs message exchange and task migration along the next dimension. In RID, an underloaded node sends load-balancing requests to only nonfaulty neighbor nodes and performs load balancing with only nonfaulty nodes; $2^n \times 100$ tasks are distributed across all nodes in an n-dimensional hypercube. The number of tasks in each node is set to 100. To model the uneven and unpredictable nature of tasks, we generated the execution time for each task as follows. The execution time for each task j in node u_i, $\tau_j(u_i)$, has a uniform distribution in the range $0 < \tau_j(u_i) < 2\tau(u_i)$: $\tau(u_i)$ is

the mean of the task execution time in node u_i; $\tau(u_i)$ has uniform distribution in the range $0 < \tau(u_i) < 2E[\tau]$. This approach was also used in [7] to model the load imbalance. Faulty nodes are generated randomly at the start of the simulation. Tasks allocated to faulty nodes are redistributed across all other nonfaulty nodes.

To investigate the effect of communication overhead on the dynamic load-balancing strategy, we assumed that the time for migration of one task is 10% of $E[\tau]$ and the time to send load information is 1% of $E[\tau]$. This assumption was also used in [5].

We compared the performance of the three dynamic load-balancing strategies on the basis of speedup over the case when no load balancing was used, i.e.,

$$Speedup = \frac{T_{\text{nobal}}}{T_{\text{bal}}},$$

where T_{bal} and T_{nobal} are execution times with and without load balancing, respectively.

Three dynamic load-balancing strategies were tested on 300 different runs of 100 tasks per node with different system sizes. Graphs of speedup versus number of faulty nodes are shown in Figs. 6–8, where the vertical bar is the 95% confidence interval computed by using a Student's t distribution. MCWA shows the best performance in the nonfaulty case. As the number of faulty nodes increases, all the strategies show a performance degradation. With a small number of faulty nodes, the MCWA algorithm performs significantly better than the alternative algorithms. However, when the number of faulty nodes increases beyond about 5, the performance of MCWA approaches that of the other methods, and sometimes even dips below the other methods. In Fig. 9, the average number of task migrations per node in the DEM and MCWA algorithms are shown for a 7-cube. As can be seen, in MCWA, the number of task migrations escalates when there are more than about four faulty nodes. This result which leads to degraded performance is due to the large tree depth of logical topologies in the MCWA algorithm when there are a large number of faulty nodes. A similar result is observed for the 5-cube and

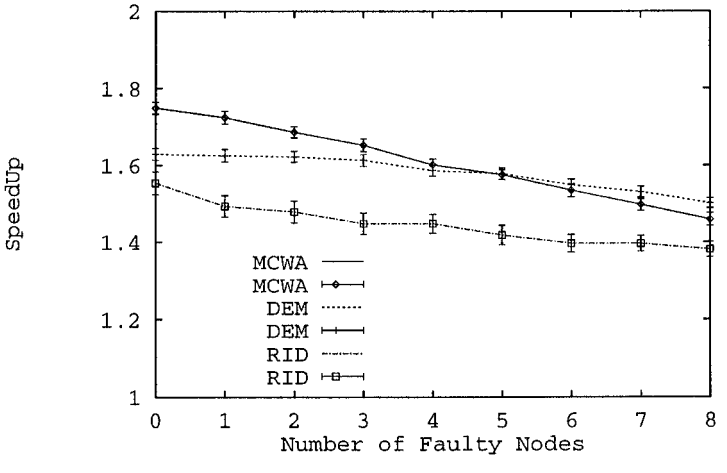

FIG. 6. Performance of MCWA, DEM, and RID in the 5-cube system.

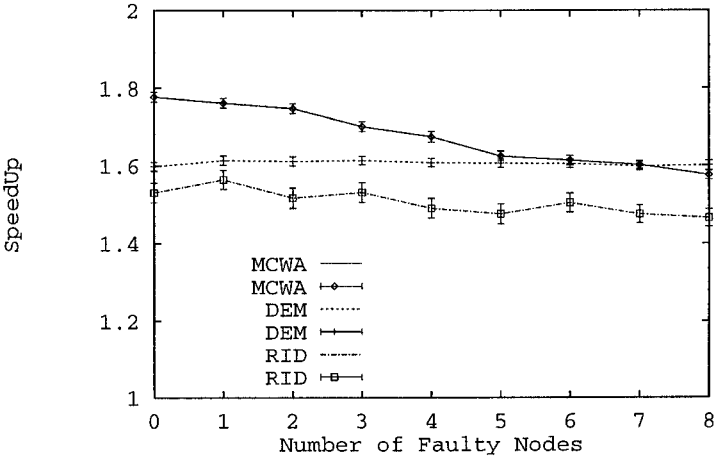

FIG. 7. Performance of MCWA, DEM, and RID in the 6-cube system.

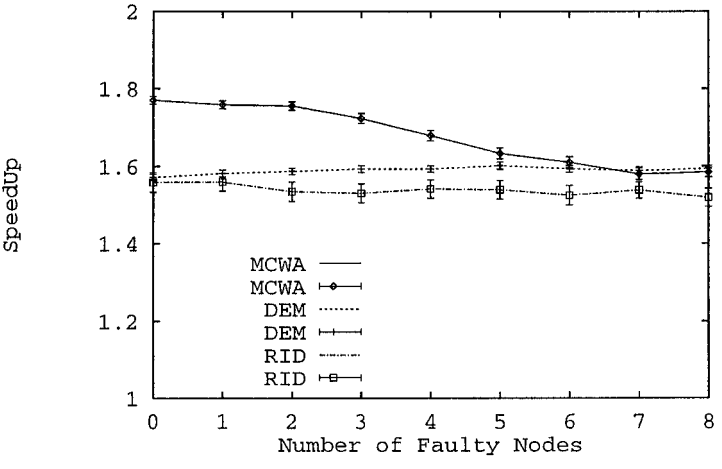

FIG. 8. Performance of MCWA, DEM, and RID in the 7-cube system.

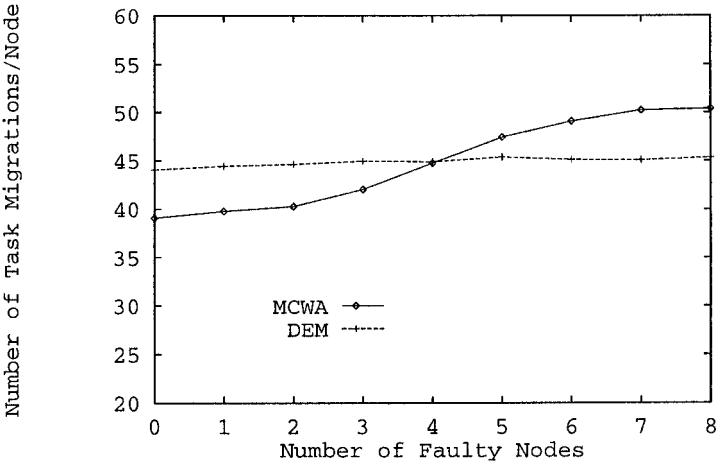

FIG. 9. Comparison of task migrations of MCWA and DEM in the 7-cube system.

6-cube—the MCWA algorithm has degraded performance in the 5-cube beyond five faults and in the 6-cube beyond six faults.

Overall, however, the relative performance of MCWA improves with the size of the hypercube, provided that the number of faulty nodes does not increase beyond the hypercube dimension. In the case of a 7-cube, the MCWA algorithm outperforms all other algorithms up to seven faulty nodes. RID shows almost constant performance irrespective of the number of faulty nodes as the RID algorithm operates independently of the topology. However, RID still performs poorly because of the absence of global load information and the many local load balancing messages. With a large number of faulty nodes, the DEM algorithm may be preferable over both the RID and MCWA algorithms.

7. DISCUSSION

In this paper, a dynamic load-balancing algorithm for hypercube multicomputers with faulty nodes has been presented, and the performance of the proposed algorithm has been evaluated through simulation. The proposed algorithm is a *synchronous dynamic* load-balancing algorithm, in which load-balancing is executed in a synchronous manner when initiated by a severely underloaded node. Such synchronous operation facilitates the accumulation of global load information through message exchanges, and also it facilitates the actual transfer of overloaded tasks.

The proposed load-balancing algorithm, denoted as MCWA, was obtained by modifying a previous synchronous dynamic load-balancing algorithm, called CWA (cube walking algorithm), to permit efficient operation in the presence of faulty nodes. A novel concept referred to as a *balancing subcube* was used to permit efficient message exchange in hypercubes with faulty nodes. MCWA was compared to another algorithm, termed DEM (dimension exchange method), for performing synchronous dynamic load-balancing in possibly faulty hypercube multicomputers, and the well-known RID (receiver initiated diffusion) algorithm. MCWA was shown to significantly outperform both DEM and RID when there are a small number of faulty nodes (less than the dimension of the hypercube). However, when the number of faulty nodes was increased beyond the dimension of the hypercube, DEM resulted in the best performance. Overall, the performance of MCWA, relative to DEM and RID, improves with the size of the hypercube, provided that the number of faulty nodes does not increase beyond the hypercube dimension.

REFERENCES

1. C. Xu and C. M. Lau, The generalized dimension exchanged method for load-balancing in k-ary n-cubes and variants, *J. Parallel and Distrib. Comput.* **24**, 1 (Jan 1995), 72–85.
2. G. Cybenko, Dynamic load-balancing for distributed memory multicomputers, *J. Parallel Distrib. Comput.* **7**, 2 (Oct. 1989), 279–301.
3. J. Kim, H. Lee, and S. Lee, Replicated process allocation for load distribution in fault-tolerant multiprocessors, *IEEE Trans. Comput.* **46**, 4 (Apr. 1997), 499–505.
4. J. Kim and K. G. Shin, Deadlock free fault-tolerant routing in injured hypercubes, *IEEE Trans. Comput.* **42**, 9 (Sept. 1993), 1078–1088.

5. K. G. Shin and Y. Chang, Load sharing in distributed real-time system with state-change broadcasts, *IEEE Trans. Comput.* **38**, 8 (Aug. 1989), 1124–1142.
6. M. J. Berger and S. H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, *IEEE Trans. Comput.* **C-36**, 5 (May 1987), 570–580.
7. M. Willebeek-Lemair and A. P. Reeves, Strategies for dynamic load-balancing on highly parallel computers, *IEEE Trans. Parallel Distrib. Systems* **4**, 9 (Sept. 1993), 979–993.
8. M. Wu and W. Shu, A load-balancing algorithm for *n*-cubes, *in* "Proc. 1996 International Conference on Parallel Processing," IEEE Computer Society, 1996, pp. 148–155.
9. N. G. Shivaratri and P. Krueger, Load distributing for locally distributed systems, *IEEE Comput.* (Dec. 1992), 33–44.
10. Y. Chang and K. G. Shin, Load sharing in hypercube-connected multicomputers in the presence of node failures, *IEEE Trans. Comput.* **45**, 10 (Oct. 1996), 1203–1211.

KYUNGWAN NAM is currently a Ph.D. student in the Department of Electrical Engineering at the Pohang University of Science and Technology (POSTECH), Pohang, Korea. He received the B.S. degree in electronic engineering from Chungang University, Seoul, Korea, in 1994, and the M.S. degree in electrical engineering from POSTECH, Pohang, Korea, in 1996. His research interests include load-balancing, high-performance computing, and network of workstations (NOW).

JAEWON SEO is currently an electrical engineer employed by Samsung Electronics. Previously, he was a Masters student in the Department of Electrical Engineering at the Pohang University of Science and Technology (POSTECH), Pohang, Korea, during his work on this paper.

SUNGGU LEE received the B.S.E.E. degree with highest distinction from the University of Kansas, Lawrence, Kansas in 1985 and the M.S.E. and Ph.D. degrees from the University of Michigan, Ann Arbor, Michigan in 1987 and 1990, respectively. He is currently an associate professor in the Department of Electrical Engineering at the Pohang University of Science and Technology (POSTECH), Pohang, Korea. His research interests are in parallel and fault-tolerant computing. Currently, his main research focus is on the high-level and low-level aspects of inter-processor communications for parallel computers.

JONG KIM received the B.S. degree in electronic engineering from Hanyang University, Seoul, Korea, in 1981, the M.S. degree in computer science from the Korea Advanced Institute of Science and Technology, Seoul, Korea, in 1983, and the Ph.D. degree in computer engineering from Pennsylvania State University, in 1991. Since 1992, he has been an assistant professor and then an associate professor in the Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Pohang, Korea. From 1991 to 1992, he was a research fellow in the Real-Time Computing Laboratory of the Department of Electrical Engineering and Computer Science, University of Michigan. His major areas of interest are fault-tolerant computing, performance evaluation, and parallel and distributed computing.