A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers

Po-Jen Chuang, *Student Member, IEEE*, and Nian-Feng Tzeng, *Member, IEEE*

***Abstract*— Fully recognizing various subcubes in a hypercube computer efficiently is nontrivial due to the specific structure of the hypercube. We propose a method with much less complexity than the multiple-GC strategy in generating the search space, while achieving complete subcube recognition. This method is referred to as a *dynamic* processor allocation scheme because the search space generated is dependent upon the dimension of the requested subcube dynamically, rather than being predetermined and fixed. The basic idea of this strategy lies in collapsing the binary tree representations of a hypercube successively so that the nodes which form a subcube but are distant would be brought close to each other for recognition. The strategy can be implemented efficiently by using right rotating operations on the notations of the sets of subcubes corresponding to the nodes at a certain level of binary tree representations. Extensive simulation runs are carried out to collect experimental performance measures of interest of different allocation strategies, and the simulated results are discussed. It is shown from analytic and experimental results that this strategy compares favorably in most of the situations with any other known allocation scheme capable of achieving complete subcube recognition.**

***Index Terms*— Binary trees, buddy strategy, hypercube computers, performance measures, processor allocation, right rotating operations.**

I. INTRODUCTION

DUE to its numerous interesting properties [1], the hypercube topology has been drawing considerable attention from researchers of different fields in recent years. Based on this topology, many prototypes and commercial systems have been built or marketed, e.g., Cosmic Cube [2], Intel iPSC/1 [3] and iPSC/2, Connection Machine [4], Ametek S/14 [5], N-Cube/10 [6], the FPS T-Series [7], and the MARK-III prototype research machine [8]. They exhibit a high potential for the parallel execution of various algorithms, and may deliver as good performance as typical supercomputers at a much lower cost.

Considered in this paper is a hypercube system where the resources are the processor nodes forming subcubes of various sizes; an incoming task incident on the system is analyzed for decomposability and the number of processors required for the task is determined. For an efficient management of processor allocation, a subcube, instead of an arbitrary number of processors, is allocated to the task, even though the number of processors actually required may be arbitrary. Specifically, a k-dimensional subcube is allocated to the task that requires p processors, where 2^k is the smallest integer such that $p \leq 2^k$. The job of a processor allocator is to find a free subcube with a size just sufficient to meet the need of the task.

The recognition of various subcubes in a hypercube system using the buddy strategy is simple and straightforward, but tends to result in poor subcube recognition ability. Later, a modified buddy strategy [12] is proposed to enhance subcube recognition ability. Chen and Shin [9] proposed the Gray Code (GC) strategy to allocate processors in a hypercube. It is shown that, compared with the buddy strategy, the GC strategy not only has a better subcube recognition ability but also reduces the fragmentation to some extent. They illustrated that the subcube recognition ability using a single GC would be twice as much as that of the buddy strategy [9]. They also proposed the use of multiple GC's to achieve full recognition of all the subcubes of any sizes. It is proved that $C(n, \lfloor n/2 \rfloor)$ GC's are sufficient to accomplish complete subcube recognition (i.e., to recognize all possible subcubes) in an n-cube, where $C(n, l)$ is the combination of n taking l at a time.

The method of determining various GC's required is, however, fairly involved so that the GC's have to be predetermined off-line and incorporated in the processor allocation algorithm [9]. If, due to some reason (such as system reconfiguration after some nodes become faulty), the size of the hypercube changes, one has to redetermine the GC's. Moreover, these $C(n, \lfloor n/2 \rfloor)$ GC's are determined independent of the required subcube dimension, k, and in the worst case, a total number of $C(n, \lfloor n/2 \rfloor)$ GC's should be searched. That is, when this multiple-GC strategy is used, the entire search space is predetermined and fixed, independent of the dimension of the subcube needed by an incoming task (thus, termed a *static* processor allocation scheme).

Another processor allocation scheme, called the free-list strategy, is introduced recently [13]. It maintains lists of free subcubes, with one list for one dimension. A requested subcube is allocated from a free list of the available subcubes. The allocation process is simple, but the deallocation procedure is very complicated. Like the multiple-GC strategy, this scheme can fully recognize all subcubes.

Here, we give a method with much less complexity than the multiple-GC strategy in generating the search space, while still achieving complete subcube recognition. The search space so generated is determined not only by n (for an n-cube) but also by k (the dimension of the subcube needed by an incoming task). Additionally, our method compares favorably in most

Manuscript received May 29, 1990. This work was supported in part by the NSF under Grant MIP-8807761 and by the Louisiana Board of Regents' R&D Program Component under Grant 86-USL(2)-127-03.

The authors are with the The Center for Advanced Computer Studies, The University of Southwestern Louisiana, Lafayette, LA 70504.

IEEE Log Number 9105062.

0018-9340/92$03.00 © 1992 IEEE

situations to other allocation schemes in terms of the mean search time, an experimental performance measure. The basic idea of this method is an extension of the buddy strategy [10]. We observe that, due to the specific interconnection pattern in a hypercube, the buddy system fails to recognize all the subcubes. In other words, the nodes that are actually "buddies" (form a subcube) have been "estranged" in the classical buddy approach. Hence, we propose to "reunite" the otherwise "estranged" buddies by "collapsing" the binary tree representation of the hypercube so that the nodes which form a subcube but are distant would be brought close to each other. It will be shown that by repeating the process of collapsing subtrees of the binary tree representation, one can bring about reunion of all estranged buddies for recognition. Since its search space is not predetermined and is dynamically changed according to k, this method is referred to as a *dynamic* processor allocation scheme.

This paper is organized as follows. Section II gives useful nomenclature and related work on hypercube processor allocation. Section III presents the proposed allocation strategy based on tree collapsing. An efficient implementation of the proposed strategy is introduced in Section IV. Section V compares the multiple-GC strategy and the proposed strategy in terms of the worst case storage and time complexities. Also provided and discussed in this section are the simulated performance measures of interest, such as the mean search time for one allocation attempt, the completion time of a sequence of allocation, and hypercube processor utilization of different allocation strategies. Concluding remarks are given in Section VI.

II. Nomenclature and Related Work

An n-dimensional hypercube, denoted by H_n, comprises 2^n nodes, each with n connection links serving as communication channels directly to n immediate neighbors. Nodes in H_n are numbered from 0 to $2^n - 1$, by n-bit binary numbers $(x_{n-1} \cdots x_i \cdots x_0)$, as their addresses. For convenience, x_i is referred to as the ith direction of the binary representation of an address. A node and any of its immediate neighbors have exactly one bit different in their addresses. A k-dimensional subcube in H_n, denoted by S_k, comprises 2^k nodes such that all the nodes have exactly $n-k$ bits identical in their addresses, with the rest k bits being *don't care*. Each address bit of a subcube is denoted by 0, 1 or *, where * is a *don't care* symbol. For example, the notation for a two-dimensional subcube in H_5 that consists of nodes 10001, 10011, 11001, and 11011 is 1*0*1. It is easy to derive that the number of distinct S_k's in H_n is $C(n,k) \times 2^{n-k}$.

The levels of an n-level binary tree are numbered from 0 to $n-1$, with the root node at level 0 and the leaf nodes at level $n-1$. Definitions 1 and 2 given below facilitate subsequent explanation.

Definition 1: A *binary tree representation* of H_n, denoted by T_n, is an $(n+1)$-level binary tree with parameter $\langle t_0, t_1, \cdots, t_{n-1} \rangle$ such that each leaf node represents a cube processor whose address comprises such a bit string that its $(n-1-t_i)$th direction, $0 \le i \le n-1$, is the ith bit of the bit sequence along the path from the root to the node (i.e., the ith parameter element, t_i, specifies the $(n-1-t_i)$th direction of the binary representation of the processor address), where $\langle t_0, t_1, \cdots, t_{n-1} \rangle$ is a permutation of $\{0, 1, \cdots, n-1\}$.

Each nonleaf node, say node A, at level l (> 0) of T_n is similarly numbered, with its l defined directions being the bit sequence along the path from the root to node A and the other $n-l$ directions being *don't care*. Apparently, node A corresponds to a subcube of dimension $n-l$, i.e., the subcube which contains such 2^{n-l} leaf nodes that have node A as their common ancestor. A binary tree representation is uniquely characterized by its parameter, so T_n with parameter $\langle t_0, t_1, \cdots, t_{n-1} \rangle$ is denoted by $T_n\langle t_0, t_1, \cdots, t_{n-1} \rangle$ henceforth. A binary tree representation of H_4, $T_4\langle 0, 2, 3, 1 \rangle$, is shown in Fig. 1, where the parameter elements denote, respectively, the 3rd, 1st, 0th, and 2nd directions of the binary representation of processor addresses. The leftmost two nodes at level 2 correspond, respectively, to the two-dimensional subcubes 0*0* and 0*1*. In general, for a given $T_n\langle t_0, t_1, \cdots, t_{n-1} \rangle$, if the address of subcube S_k associated with a node at level $n-k$ is $x_{n-1}x_{n-2} \cdots x_j \cdots x_0$, then x_j is a *don't care* bit if $j = n-1-t_i$, with i satisfying $n-k \le i \le n-1$. For example, the addresses of S_4's associated with the nodes at level 2 of $T_6\langle 1, 3, 4, 5, 0, 2 \rangle$ are $*x_4*x_2**$, where $x_4x_2 \in \{00, 01, 10, 11\}$.

Clearly, there are $n!$ distinct binary tree representations of H_n, since there are $n!$ distinct parameters which are the permutations of $\{0, 1, \cdots, n-1\}$. Without an appropriate procedure to determine the search space, one would have to exhaustively examine all these $n!$ trees in order to guarantee complete subcube recognition, yielding an intractable situation.

Definition 2: The *primary binary tree representation* of H_n, denoted by PT_n, is $T_n\langle 0, 1, 2, \cdots, n-1 \rangle$, i.e., the leaf nodes of PT_n have addresses 0 through $2^n - 1$, starting from the leftmost node toward the rightmost one.

Fig. 2 gives an example of PT_4.

The processors in H_n must be allocated to incident tasks in such a way that processor utilization is maximized (or equivalently, a task is assigned an available subcube whenever such a subcube exists). Due to the special structure of a hypercube, it is nontrivial to detect the availability of a subcube. To accomplish this, ways of exploiting the ideas used in conventional memory allocation approaches have been focused, giving rise to two major types of strategies—the buddy strategies and the Gray code strategies. The former type includes the simple buddy strategy and the modified buddy strategy [12], which are based on the buddy system [10]; whereas the latter type contains the single Gray code strategy and the multiple Gray codes strategy [9]. Any of these strategies performs a linear search on a list of allocation bits by means of the first-fit search process. Recently, a free-list strategy is proposed [13], where a separate list is used for keeping all the free subcubes of a certain dimension. In what follows, we briefly review these strategies.

A. The Buddy and Modified Buddy Strategies

Since H_n has 2^n processors, it requires 2^n allocation bits to keep track of the availability of all the processors. An

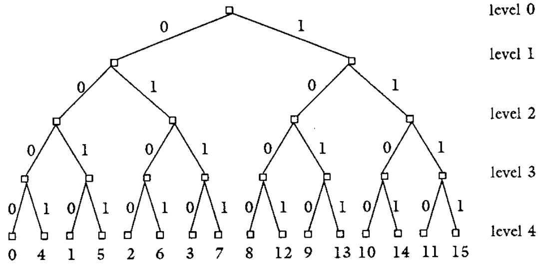


Fig. 1. A binary tree representation (T_4) with a parameter $\langle t_0, t_1, t_2, t_3 \rangle = \langle 0, 2, 3, 1 \rangle$.

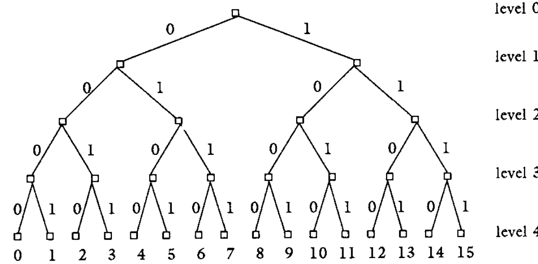


Fig. 2. The primary binary tree representation of H_4.

allocation bit with 0 (or 1) indicates that the corresponding processor is available (or unavailable).

The buddy strategy [9] can be easily explained by the primary binary tree representation of H_n. When an incoming task needs an S_k, this strategy searches level $n-k$ of the primary binary tree from left to right and allocates the first available subcube (associated with a node in the level) to the task. Clearly, only $2^{n-k} S_k$'s can be recognized, where $1 \le k \le n-1$. The strategy is proved to be optimal in the sense that it can always accommodate any input task sequence $\{I_i\}_{i=1}^k$ if and only if $\sum_{i=1}^k 2^{|I_i|} \le 2^n$, where $|I_i|$ is the dimension of a subcube required to accommodate task I_i.

The modified buddy strategy [12] can also be illustrated by the primary binary tree representation of H_n. The ith partner of a node at level l (> 0) $x_{n-1}x_{n-2}\cdots x_{i+1}x_i x_{i-1}\cdots x_{n-l}$ for any $n-l \le i \le n-1$ is defined as the node $x_{n-1}x_{n-2}\cdots x_{i+1}1x_{i-1}\cdots x_{n-l}$ if $x_i = 0$, or undefined if $x_i = 1$ (note that the $n-l$ tailing *don't care* bits in the binary address representation of the node are disregarded). The nearest partner of a node is the ith partner of the node, where i is the smallest integer. A node is free if and only if all of its descendants are free. When a k-dimensional subcube is requested, this strategy searches level $n-k+1$ of the primary binary tree, from left to right, for the first free node and its nearest free partner. If found, it allocates the subcube formed by the two $(k-1)$-dimensional subcubes corresponding to the two free nodes. This strategy is shown to be able to recognize $(n-k+1) \times 2^{n-k}$ distinct S_k's [12]. An extension to the strategy is also provided: if two free nodes forming an available S_k are unavailable at level $n-k+1$, the higher levels are searched to get more S_k's. Totally, as many as $(k \times (n-k)+1) \times 2^{n-k}$ distinct S_k's can then be recognized.

B. The GC and Multiple-GC Strategies

A GC is a sequence of binary numbers where any two successive numbers have only one different bit. The Chen and Shin's allocation strategy [9] is based on the binary reflected Gray Code (BRGC), the best known GC. The strategy is similar to the buddy strategy except that the allocation bits are stored in the BRGC sequence.

The GC strategy [9] can be explained by a binary tree with the addresses of leaf nodes denoted by BRGC. According to the property of BRGC, any two adjacent nodes (including the leftmost and rightmost pair, which enables a circular search) in the $(n-k+1)$th level form an S_k even if they may not have the same immediate predecessor. When an S_k is requested, this strategy searches from left to right for *two* available adjacent nodes in *level* $n-k+1$ rather than for *an* available node in *level* $n-k$. Therefore, it recognizes twice as many subcubes as the buddy strategy.

Since a GC cannot recognize all the subcubes in a hypercube and different GC's recognize different sets of subcubes, Chen and Shin have proved that, for complete subcube recognition, $C(n, \lfloor n/2 \rfloor)$ GC's are needed and sufficient [9]. Fully recognizing all the subcubes of any dimension is unachievable unless $C(n, \lfloor n/2 \rfloor)$ GC's are employed. Although a method is devised to identify such GC's, the process is so complicated that it has to be done off-line in practical implementation, as suggested in [9]. Besides, storage needed for keeping these GC's is quite large.

C. The Free-List Strategy

This strategy maintains lists of free subcubes available in the hypercube, with one list for a dimension [13]. An incoming request for dimension k gets allocated by assigning the first element in the free list of dimension k, if the list is not empty; otherwise, by decomposing an available subcube with dimension $> k$. Although the allocation steps are simple, the strategy involves a quite complicated deallocation process whenever a subcube is released, in order to guarantee its correctness. Specifically, the deallocation process has to: 1) merge the released subcube with any other subcube to form a bigger cube, or generate another available cube of the same dimension; 2) search all subcubes of the newly produced cube and remove them from their corresponding lists; and 3) repeat the first two steps until nothing can be done further.

Although this strategy assures full subcube recognition, its time and storage complexities in the worst case can be very high, and will be much higher than $O(n^3)$ claimed in [13] for H_n, because the list for dimension k can grow as much as $O(2^{n-k})$. (The claimed complexity is derived by assuming the list length to be $O(n)$, which is observed from simulation under light traffic.)

III. Allocation Strategy Using Tree Collapsing

So far, only two allocation strategies achieve complete subcube recognition, namely, the multiple-GC strategy and the free-list strategy. In the following, we introduce a new recognition-complete allocation strategy based on tree collapsing, termed the *TC strategy,* which exhibits better per-

formance in most situations as compared to the other two recognition-complete allocation strategies. Before presenting the TC strategy, we give a definition.

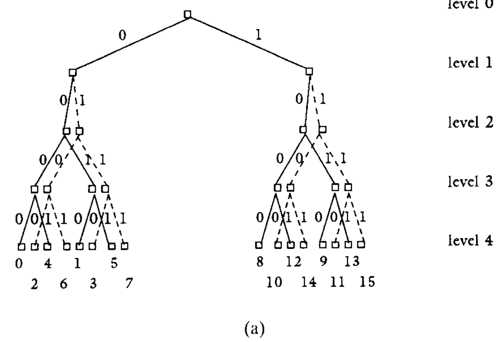
Definition 3: A *Collapsing Transform (C-transform)* is a transform which operates on a binary tree representation of H_n (i.e., T_n) and produces a *collapsed tree*. A C-transform at level i, denoted by ξ_i, involves i) collapsing T_n such that the left and right subtrees of every node at level i are clustered together without changing their relative locations (the left descendant stays left, while the right one stays right) and then ii) swapping the incoming links of the two inner nodes in every block of four nodes at level $i+2$ throughout level n.

The result of performing ξ_1 on $T_4\langle 0,2,3,1\rangle$ of Fig. 1 is shown in Fig. 3. Fig. 3(a) illustrates the tree collapsing at level 1 after i), and then the incoming links of leaf nodes 2 and 4, 3 and 5, etc., are swapped, so are those of the two inner nodes in every block of four nodes at level 3, yielding the collapsed tree $T_4\langle 0,3,1,2\rangle$ given in Fig. 3(b). From the way the collapsed trees are generated, we have the following lemma immediately.

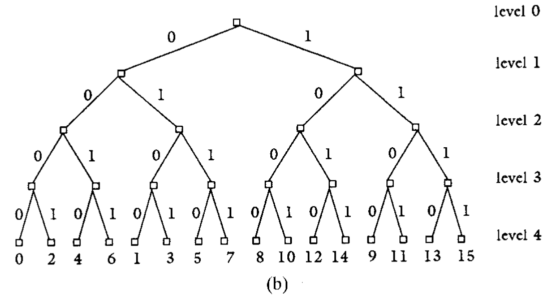
Lemma 1: Given $T_n\langle t_0, t_1, \cdots, t_{n-1}\rangle$, if the parameter of the collapsed tree T'_n derived by applying ξ_j, $0 \le j \le n-1$, to T_n is $\langle t'_0, t'_1, \cdots, t'_{n-1}\rangle$, then

1) $t'_i = t_i$, for $0 \le i < j$,
2) $t'_i = t_{i+1}$, for $j \le i < n-1$, and
3) $t'_{n-1} = t_j$.

Actually, $\langle t'_0, t'_1, \cdots, t'_{n-1}\rangle$ results from $\langle t_0, t_1, \cdots, t_{n-1}\rangle$ by operating left rotation on t_i, where $j \le i \le n-1$ for a certain j. As an instance, the parameters of the collapsed tree derived from applying ξ_1 to $T_6\langle 1,3,4,5,0,2\rangle$ are $\langle 1,4,5,0,2,3\rangle$.



(a)



(b)

Fig. 3. A C-transform ξ_1 on T_4 of Fig. 1. (a) The collapsing of T_4 at level 1 after i), and (b) the collapsed tree—another binary tree representation with $\langle t_0, t_1, t_2, t_3\rangle = \langle 0,3,1,2\rangle$.

A. Algorithm for Collapsed Trees Generation

Our allocation approach is realized by recognizing the fact that in the buddy strategy the nodes forming subcubes are not adjacent to each other, and only when they are brought adjacent to each other would complete subcube recognition become achievable. The following algorithm performs C-transforms successively on binary tree representations, starting with the primary one until nodes of every possible subcube of the desired dimension are brought together in one of the tree representations. It serves as the basis of the TC strategy.

Algorithm G

Input: the primary binary tree representation (stored in primary_tree) of H_n and the dimension of a subcube to be allocated, k, where primary_tree, new_tree, pre_tree are binary tree data structures.

```
Procedure main(n, k, primary_tree);
begin
   if (k = 0 or k = n) then stop;
   for i := n − k − 1 downto 0
      call collapse(i, primary_tree, new_tree);
      call subsequent_collapse(i, 0, new_tree);
   endfor
end;
Procedure subsequent_collapse(pre_level,
   pre_step, pre_tree);
begin
   step := pre_step+1; /* advancing one step */
   if step ≥ k then return
   else
      for i := pre_level to n − k − 1
         call collapse(i, pre_tree, new_tree);
         call subsequent_collapse(i, step, new_tree);
      endfor
   endif
end
```

There is no need for performing the C-transform when $k = 0$ or $k = n$. The routine collapse(i, old_tree, new_tree) performs ξ_i on old_tree and stores the resultant collapsed tree in new_tree; whereas the routine subsequent_collapse(pre_level, pre_step, new_tree) generates subsequent collapsed trees recursively from new_tree by using the collapse routine. According to Algorithm G, the possible C-transforms for a given n and k are depicted in Fig. 4, where the C-transforms in one column are performed by one step of Algorithm G and totally there are k steps (from step 0 to step $k-1$). The numbers of C-transforms in Fig. 4 counted step by step are given in Fig. 5.

Definition 4: Two binary tree representations of H_n, T_n and T'_n, are *overlapped at level* $n-k$ if there exist k-dimensional subcubes S_k and S'_k associated, respectively, with the nodes at levels $n-k$ of T_n and T'_n such that $S_k = S'_k$. Otherwise, T_n and T'_n are *nonoverlapped at level* $n-k$.

Theorem 1: In Algorithm G, i) the total number of C-transforms performed is $C(n,k)-1$, and ii) the primary and collapsed trees are pairwise nonoverlapped at level $n-k$.

A proof of this theorem can be found in Appendix. The result of performing Algorithm G on the case of $n = 4$ and

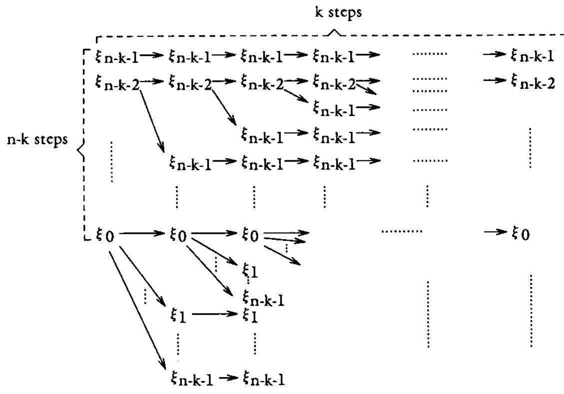


Fig. 4. The possible C-transforms performed in Algorithm G for any given n and k.

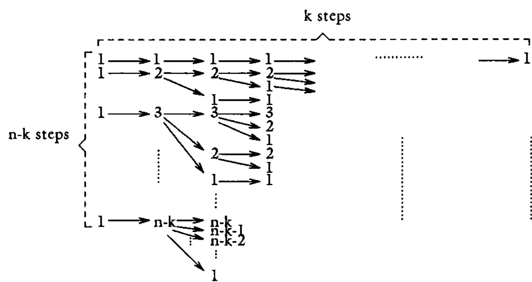
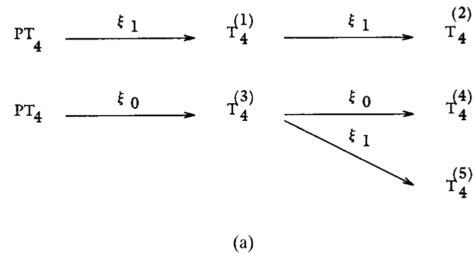


Fig. 5. The numbers of C-tansforms in Fig. 4 (counted step by step).



(a)

PT_4 :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$T_4^{(1)}$:	0	4	1	5	2	6	3	7	8	12	9	13	10	14	11	15
$T_4^{(2)}$:	0	2	4	6	1	3	5	7	8	10	12	14	9	11	13	15
$T_4^{(3)}$:	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7	15
$T_4^{(4)}$:	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
$T_4^{(5)}$:	0	2	8	10	1	3	9	11	4	6	12	14	5	7	13	15

(b)

Fig. 6. All C-transforms performed by Algorithm G for $n = 4$ and $k = 2$. (b) The leaf nodes of collapsed trees. (The parenthesized numbers indicate the collapsing order.)

$k = 2$ is given in Fig. 6. The number of produced trees is 5, which equals $C(4, 2) - 1$, and each four leaf nodes associated with any node at level 2 of PT_4 or of any produced T_4 form a distinct S_2.

B. The Tree Collapsing Strategy

From Theorem 1, it is easy to derive that all the nodes at level $n - k$ of PT_n and of all produced T_n's correspond to $C(n, k) \times 2^{n-k}$ distinct S_k's (which are all the possible distinct subcubes with dimension k). If the buddy strategy is applied to *all* of these binary tree representations, fully recognizing all S_k's can be achieved. Therefore, we propose the following tree collapsing strategy.

Processor Allocation:

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

Step 2. Search level $n - k$ of the primary binary tree from left to right and find the first available subcube (the allocation bits of the nodes of the subcube, i.e., the leaf nodes corresponding to a node at level $n - k$, are all 0's), if any, then go to Step 4.

Step 3. Perform Algorithm G and search level $n-k$ of one produced collapsed tree (if any) at a time, from left to right, until the first available subcube is found. If an available subcube is found, go to the next step; otherwise, go to Step 5.

Step 4. Set the corresponding allocation bits of the available subcube to 1's, and then allocate the subcube to task I_j. Stop.

Step 5. Attach I_j to the task queue and wait until a subcube is released.

Processor Relinquishment: Reset those allocation bits of the released subcube to 0's.

The proposed TC strategy can fully recognize all subcubes of any given dimension. It is a dynamic processor allocation strategy, as its search space varies when the subcube dimension changes. Apparently, the TC strategy is also optimal in the sense that it can always accommodate any input task sequence $\{I_i\}_{i=1}^{k}$ if and only if $\sum_{i=1}^{k} 2^{|I_i|} \le 2^n$, where $|I_i|$ is the dimension of a subcube required to accommodate task I_i, as in the buddy strategy mentioned earlier.

IV. Efficient Implementation of the TC Strategy

A direct implementation of the TC strategy needs a large store for keeping binary tree structures and a long execution time for performing C-transforms. To reduce the complexity, an implementation based on the perfect shuffle operation [14] has been proposed recently [15]. In the following, we introduce a more efficient new implementation which is realized by means of the *right rotating* operation.

A. An Implementation

Let a set of subcubes S_k's corresponding to the 2^{n-k} nodes at level $n - k$ of $T_n\langle t_0, t_1, \cdots, t_{n-1}\rangle$ be denoted by $\Phi_{S_k}^{T_n\langle t_0, t_1, \cdots, t_n\rangle}$. $\Phi_{S_k}^{T_n\langle t_0, t_1, \cdots, t_n\rangle}$ can be expressed by an n-bit representation (called a *notation*) where each representation bit is indicated by either □ (a *select* symbol) or * (a *don't care* symbol). Since there are exactly k don't care bits in the address of any S_k, the notation of $\Phi_{S_k}^{T_n\langle t_0, t_1, \cdots, t_n\rangle}$ thus

consists of k don't care bits and $n-k$ select bits. According to Definition 1, it is apparent that the k don't care bits are in directions $n-1-t_{n-k}, n-1-t_{n-k+1}, \cdots$, and $n-1-t_{n-1}$, with the other bits being select bits. It is easy to derive that if $a_0 a_1 \cdots a_{n-1}$ represents $\Phi_{S_k}^{T_n\langle t_0,t_1,\cdots,t_n\rangle}$, then $a_i = *$ if $i \in \{t_{n-k}, t_{n-k+1}, \cdots, t_{n-1}\}$; otherwise, $a_i = \Box$. Consequently, any $\Phi_{S_k}^{T_n\langle t_0,t_1,\cdots,t_n\rangle}$ has a unique n-bit notation; whereas any n-bit notation with k don't care bits represents a unique set of 2^{n-k} subcubes S_k's.

The address of any S_k in the subcube set $\Phi_{S_k}^{T_n\langle t_0,t_1,\cdots,t_n\rangle}$ can be obtained from the set's notation with an appropriate value, 0 or 1, chosen for each of its $n-k$ select bits. As there are totally 2^{n-k} ways to choose appropriate values for those select bits, the set $\Phi_{S_k}^{T_n\langle t_0,t_1,\cdots,t_n\rangle}$ contains exactly the 2^{n-k} distinct subcubes corresponding to the 2^{n-k} nodes at level $n-k$ of $T_n\langle t_0, t_1, \cdots, t_{n-1}\rangle$. It is obvious that the leftmost $n-k$ bits and the rightmost k bits of the notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$ (which is the set of S_k's corresponding to the 2^{n-k} nodes at level $n-k$ of PT_n, the primary binary tree representation) are the select bits and don't care bits, respectively.

Any C-transform, say ξ_i, in Fig. 4 transforms a binary tree, say $T_n\langle t_0, t_1, \cdots, t_{n-1}\rangle$, into a collapsed tree, say $T'_n\langle t'_0, t'_1, \cdots, t'_{n-1}\rangle$, and the sets of S_k's corresponding to the 2^{n-k} nodes at level $n-k$ of T_n and of T'_n can be represented by two unique notations, say γ and γ', respectively. According to Lemma 1, parameter $\langle t'_0, t'_1, \cdots, t'_{n-1}\rangle$ can be obtained from $\langle t_0, t_1, \cdots, t_{n-1}\rangle$ by operating left rotation on t_j, where $i \le j \le n-1$. Since a notation can produce the addresses of all its involved subcubes easily, it appears possible to implement our TC strategy more efficiently than directly performing C-transforms on binary tree representations. To this end, we have to know through what operation notation γ' can be derived from γ, or in general, to identify a simple operation capable of generating all distinct notations systematically. Before stating such a desired operation, we first give the following definition.

Definition 5: A *Right Rotating Transform (R-transform)* is a transform which operates on an n-bit notation, say $A = a_0 a_1 \cdots a_{n-1}$. An R-transform from bit i, denoted by ρ_i, performs right rotation on the rightmost $n-i$ bits of A and produces a *rotated notation* $A' = a'_0 a'_1 \cdots a'_{n-1}$ such that

1) $a'_j = a_j$, for $0 \le j < i$,
2) $a'_j = a_{j-1}$, for $i < j \le n-1$, and
3) $a'_i = a_{n-1}$.

To give an example, the rotated notation of performing ρ_2 on a 6-bit notation □*□*□* is □**□*□, as shown in Fig. 7.

The sequences of C-transforms carried out in Algorithm G for any given n and k, exhibited in Fig. 4, show that, for each sequence, C-transforms always perform on nodes earlier at a lower level than at a higher level, with the highest level not exceeding $n-k-1$, and the total number of C-transforms is k. Each of such C-transforms is found to have a corresponding R-transform, as demonstrated in the following theorem.

Theorem 2: Let $\xi_{i_0}, \xi_{i_1}, \cdots, \xi_{i_{k-1}}$ be a sequence of C-transforms applied to PT_n (where $i_p \ge i_q$ if $p > q$ and $0 \le p, q \le k-1$), and $\rho_{i'_0}, \rho_{i'_1}, \cdots, \rho_{i'_{k-1}}$ be a sequence of R-transforms made on the notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$. If ξ_{i_r} produces $T_n\langle t_0^{(r+1)}, t_1^{(r+1)}, \cdots, t_{n-1}^{(r+1)}\rangle$, then $\rho_{i'_r}$, $i'_r = i_r + r$, produces the notation of $\Phi_{S_k}^{T_n\langle t_0^{(r+1)}, t_1^{(r+1)}, \cdots, t_{n-1}^{(r+1)}\rangle}$, where $0 \le i_r \le n-k-1$ and $0 \le r \le k-1$.

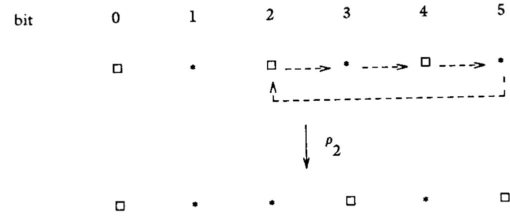


Fig. 7. An example of performing ρ_2.

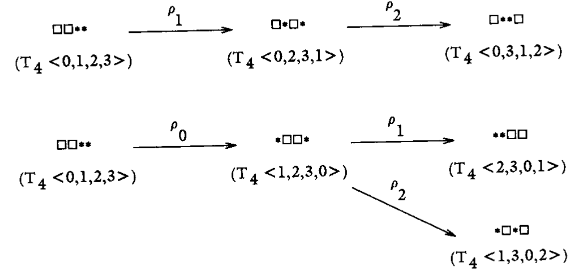


Fig. 8. All R-transforms perform by Algorithm G* for $n = 4$ and $k = 2$. (The parenthesized item below each rotated notation indicates the collapsed tree produced from the corresponding C-transform in Algorithm G.)

A proof of this theorem is provided in Appendix. According to the theorem, Algorithm G can be revised as follows to fit an efficient implementation making use of R-transforms. The binary tree representation $T_n\langle t_0, t_1, \cdots, t_{n-1}\rangle$ is replaced by the n-bit notation of $\Phi_{S_k}^{T_n\langle t_0,t_1,\cdots,t_n\rangle}$, and a ρ_{i+r} is performed each time when a ξ_i is required at step r. For example, the subroutine call collapse(i, pre_tree, new_tree) in Algorithm G would be replaced by routine rotate($i+r$, pre_notation, new_notation) at step r, which performs ρ_{i+r} on the notation of $\Phi_{S_k}^{T_n\langle t_0,t_1,\cdots,t_n\rangle}$, i.e., pre_notation, to get a rotated notation new_notation for pre_tree = $T_n\langle t_0, t_1, \cdots, t_{n-1}\rangle$. The revised Algorithm G is referred to as Algorithm G* in our later discussion. The result of performing Algorithm G* on the case of $n = 4$ and $k = 2$ is depicted in Fig. 8. Compared with Fig. 6, the above theorem is readily verified.

To implement the TC strategy by using R-transforms, initially an n-bit notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$ (rather than the primary binary tree $(T_n\langle 0, 1, \cdots, n-1\rangle)$ itself) is given. Certain steps in the TC strategy described in the previous section need to be modified accordingly. Instead of searching level $n-k$ of the primary binary tree and the collapsed trees for an available subcube (at Step 2 and Step 3 of the allocation procedure stated in Section III-B), we now search the sets of subcubes dictated by the initial notation and the rotated notations in order to find the first free subcube. Also, Algorithm G performed at Step 3 is substituted by Algorithm G*.

B. Implementation Considerations

The TC strategy can be carried out either centrally by a host

processor or in a distributed manner using several processors, as explained below.

Centralized Allocation Scheme: In a centralized scheme, a copy of 2^n allocation bits (which initially are all 0's) is kept in the host processor, and only the host processor is responsible for allocation. (An allocation bit being 0 (or 1) means that its corresponding processor is available (or unavailable).) When a request for an S_k is made, the host processor starts to generate the n-bit notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$ and then search the allocation bits of the processors in each subcube $\in \Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$ one by one. (Note that "0" and "1" can be used to represent the select symbol and the don't care symbol, respectively, in the implementation since only two symbols are present in the notation.) This is carried out as follows: Sequentially feed the $n-k$ select bits of the notation of the subcube set $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$ with values ranging from 0 to $2^{n-k}-1$ in any $(n-k)$-bit GC sequence, so as to acquire the address of each subcube in the set one by one. Recall that two successive numbers in a GC sequence differ in exactly one bit. Choosing the values of the $n-k$ select bits in this fixed order tends to result in assigning subcubes more adjacently, reducing possible fragmentation to some extent. When the address of a subcube in the set is acquired, the allocation bits of its involved processors can be identified rapidly (by considering only the positions of its select bits). When the host processor finds out the first subcube S_k with its processors' allocation bits being all 0's, it sets those bits to 1 and assigns S_k to the request. In case no such S_k is available, Algorithm G* is performed to derive rotated notations which represent disjoint sets of subcubes. These sets of subcubes are then searched one by one in the same way until a free subcube is found and assigned to the request. If no subcube of the required size is available to a request, the request is queued until processor relinquishment makes room for it. When an S_k is released, the corresponding bits are reset to 0's.

Distributed Allocation Scheme: In a distributed scheme, multiple processors are involved in allocation since some parallelism can be extracted from Algorithm G* and executed by several processors in parallel. Requests for subcubes are managed by a host processor, which keeps the array of allocation bits. On receiving a request, the host processor supplies a copy of the allocation bits to each of those processors involved in executing the allocation program, and also provides it with information as to what sequence of R-transforms is to be carried out. Those processors then start to perform the sequences of R-transforms individually and follow the allocation procedure to find out if a required S_k is available. Each involved processor sends the result back to the host processor, indicating whether a required S_k is available (i.e., a "P" report) or not (i.e., an "N" report). The host, on receiving a "P" report from an involved processor, immediately aborts the unfinished R-transforms as well as searches being executed by the other processors and then assigns the available subcube to the incoming task. The host processor also updates the array of allocation bits accordingly. On the other hand, when χ (= the total number of involved processors) "N" reports are received, the host processor knows that there is no subcube available for the incoming task.

The processors involved in performing the allocation job can be predetermined or can be dynamically chosen by the host processor upon the arrival of each request. In the former way, processors are initially selected and dedicated solely to executing the allocation program; they are excluded from executing incoming tasks. The number of processors available for executing incoming tasks is thus reduced, leading to degraded system utilization. On the other hand, those involved processors in the latter way are chosen arbitrarily as long as their corresponding allocation bits are 0's, because they work independently and need not communicate with one another during the entire course of performing the allocation job (and it is easy to see from Fig. 4 that they need to communicate only with the host processor). On completing the allocation of a request, they are set free (by the host processor) and may participate in executing the current incoming task. The allocation job can be carried out in parallel this way without reducing the number of processors available for serving incoming tasks. However, since the processors participating in the allocation job are arbitrary and not fixed, each time the host processor needs to send to each participating processor the allocation bit array and the sequence of R-transforms to be performed. The allocation program also has to be sent over, unless it is kept in every processor of H_n (which apparently yields considerable storage overhead).

V. Performance Comparison

There are three allocation strategies able to achieve complete recognition of all subcubes, namely, the multiple-GC strategy [9], our TC strategy, and the free-list strategy [13]. Unlike the first two, the free-list strategy involves a very complicated deallocation procedure in order to guarantee the correctness of the free lists maintained, whenever a subcube is released. The overall time complexity claimed in [13] is $O(n^3)$, which is derived by assuming the list length to be $O(n)$. In fact, the claimed value reflects at best only the "observed" worst case under their simulation conditions. Consider a request for subcube S_k in H_n. Theoretically, the list length can grow as large as $O(2^{n-k})$, when only one or two lists are nonempty, say the lists for 0 or 1 dimension. This gives rise to the time complexity of at least $O(n^2 2^{n-k})$. The exact time complexity, however, is fairly difficult to estimate. As a result, we analyze only the first two strategies to derive their time and storage complexities under the worst situation. Then, the performance measures of interest of the three strategies obtained through simulation runs are provided and compared.

A. Worst Case Analysis

It is clear from [9] that the multiple-GC strategy needs storage to keep $C(n, \lfloor n/2 \rfloor)$ GC's and the arrays of allocation bits, whereas ours needs storage only to keep $k+1$ n-bit representations (i.e., the initial notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$ and the k rotated notations) and an array of allocation bits, which is far less. For example, besides the storage required to keep allocation bits, the multiple-GC strategy requires 252K words

storage, assuming one word is used for each address of 10-cube nodes, whereas the TC strategy requires no more than 11 words, one word for each 10-bit notation ($k \leq 10$).

For the multiple-GC strategy, to achieve complete subcube recognition, $C(n, \lfloor n/2 \rfloor)$ appropriate GC's must be generated off-line at the initialization step. When n (the dimension of the target hypercube) changes, the GC's should be redetermined. At each allocation, the entire search space has to be examined in case the required subcube is not found, independent of the dimension of the subcube requested. For each GC, one needs to check up to 2^{n-k+1} blocks, with each containing 2^k elements. Totally, in the worst case $C(n, \lfloor n/2 \rfloor) \times 2^{n-k+1}$ blocks in the search space must be checked, giving rise to search time complexity $C(n, \lfloor n/2 \rfloor) \times 2^{n-k+1} \times \eta(2^k)$, where $\eta(2^k)$ (a function of 2^k) is the time required for a linear search over one block of size 2^k.

For the TC strategy, no initialization step is needed, but the search space is generated each time a request comes up. Assume that each R-transform takes one unit time, then, to generate the search space requires $C(n, k) - 1$ time units, where 1 accounts for the initial notation of $\Phi_{S_k}^{T_n \langle 0,1,\cdots,n-1 \rangle}$. Since up to 2^{n-k} subcubes are searched for each notation and there are $C(n, k)$ notations in the search space, totally $C(n, k) \times 2^{n-k}$ subcubes are examined in the worst case, yielding search time $C(n, k) \times 2^{n-k} \times \eta(2^k)$.

We plot the search times of the two strategies for H_{10} as a function of k in Fig. 9, where $\eta(2^k)$ is assumed to be 2^k. It is obvious that the TC strategy always has a far less search time. The search space generation time for the TC strategy is also shown in the figure. The time for allocating an arbitrary subcube using the TC strategy is the search time plus the search space generation time (which is effectively negligible). From the figure, it is clear that the TC strategy always takes a considerably less time.

Since any subcube address is generated by an n-bit notation, it is possible to employ an "associative search device" to greatly reduce the search time. This search device would store all the processors' addresses and their corresponding allocation bits. It takes each generated subcube address and compares only the select bits of the subcube address with the corresponding bits of all the processors' addresses simultaneously, examining the availability of the subcube fast. Such a device is expected to have reasonably low complexity but is able to reduce the search time significantly (i.e., the time needed to examine the availability of a subcube is then reduced from $\eta(2^k)$ to a small constant), making the TC strategy even more advantageous.

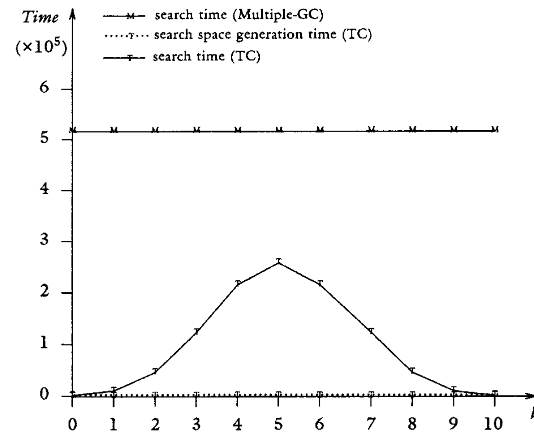


Fig. 9. Time required for allocating an S_k in H_{10} in the worst case.

B. Simulation Results

The simulation model used for comparing the Multiple-GC, free-list, and TC strategies is described below. Task allocation is carried out centrally by a dispatching processor exclusive from the simulated hypercube, so that a request for all of the hypercube nodes is allowed. Initially, the simulated hypercube is empty, and 100 tasks are generated and queued at the dispatcher. The dimensions of the subcubes requested by the 100 tasks are assumed to follow a given distribution. The residence times of allocated subcubes are assumed to have a uniform distribution. At each time unit, the dispatcher attempts to allocate the task at the top of the queue. If the dispatcher fails to identify an available subcube of the desired dimension for the task, it reattempts in the subsequent time units until a free subcube is eventually obtained, i.e., an FCFS scheduling discipline is followed. (Note that it could be done in a more efficient way if the dispatcher reattempts only when a subcube is released.) When an available subcube is allocated to the task, the task is removed from the queue and the next task is served at the following unit time. No new task is generated during the course of simulation.

Under the above simulation model, we collect such performance measures as the completion time (Ξ—the time taken to finish all the 100 tasks), processor utilization (the percentage of a processor being utilized per unit time) over the period Ξ, and the mean search time for each allocation attempt. To allocate a task in H_n, both the multiple-GC and TC strategies perform searches over the allocation (i.e., status) bits, whereas the free-list strategy[1] involves comparisons between pairs of n-character strings (which are kept in linear lists and each string element can be "0," "1," or "x"). In our simulation, we assume that the time to examine an allocation bit is the same as that to compare a pair of n-character strings, with each taking a cycle. (Note that, practically, to look up a bit may be faster.) The simulation results are averaged over 20 independent runs.

The results for task allocation in H_5 are shown in Table I. Subcube sizes requested by the 100 tasks are governed by three distributions—the uniform, normal, and decreasing distributions. The uniform distribution seems to well reflect the situations where the nature of tasks to be executed is unknown. By contrast, the normal distribution gives a higher probability for requesting subcubes whose dimensions approach the middle of the system dimension, while the decreasing distribution indicates that the probability of requesting a larger subcube is lower. (The probabilities of requesting i-dimensional subcubes, p_i, $0 \leq i \leq n$, for these two cases are shown under

[1]The procedure for free-list strategy was provided by J. Kim and C. R. Das.

TABLE I
SIMULATION RESULTS FOR TASK ALLOCATION IN H_5

Residence time distribution	Subcube size distribution	Completion time			Processor utilization			Search time			NR*
		MGC*	FL*	TC	MGC	FL	TC	MGC	FL	TC	
uniform (5,10)	uniform	366.2	366.2	366.2	66.3	66.3	66.3	63.1	208.2	24.4	2.5
	normal**	271.7	271.1	271.7	70.6	70.7	70.6	48.7	130.6	29.4	3.8
	decreasing***	159.9	159.6	159.9	58.2	58.3	58.2	96.7	372.5	27.4	1.9
uniform (20,30)	uniform	1097.7	1097.8	1097.7	74.3	74.3	74.3	63.5	220.5	24.3	2.9
	normal	817.1	814.9	817.0	78.6	78.8	78.6	49.9	122.8	31.5	4.7
	decreasing	425.0	427.2	424.9	73.6	73.1	73.6	95.7	396.0	32.5	3.0

*MGC: Multiple-GC, FL: Free-List, NR: number of right rotating operations.
**$p_0 = 0.064$, $p_1 = 0.152$, $p_2 = 0.284$, $p_3 = 0.284$, $p_4 = 0.152$, $p_5 = 0.064$.
***$p_0 = 0.488$, $p_1 = 0.202$, $p_2 = 0.132$, $p_3 = 0.092$, $p_4 = 0.054$, $p_5 = 0.032$.

the table.) The normal distribution reflects that the more the possible subcubes with a certain size, the more frequently they are requested (as there exist more possible subcubes whose dimensions approach to the middle); whereas the decreasing distribution may reflect better the cases where most tasks have reasonable parallelism but are not suitable for fine-grained partition. The residence time distribution of *uniform*(5,10) indicates that the time is uniformly distributed between 5 and 10 time units. The 10 GC's used by the multiple-GC strategy are those given in [9]. The results shown in the table are reasonably accurate. As an instance, for the first value 366.2 provided in the table, given 95% confidence, the calculated confidence interval half-width over the 20 replications is 18.1, meaning that we are 95% confident that the true result would fall into the interval 366.2 ± 18.1, or equivalently, 366.2 ± 4.9%. This simulated value has only less than 5% error.

It is interesting to observe from the table that the completion time and processor utilization are virtually identical for the three strategies, with negligible differences resulting possibly from the simulation variation. We speculated that all allocation strategies which have the complete subcube recognition ability would lead to pretty much identical processor utilization (and hence, the same completion time). The speculation is supported by the simulated data provided in [13], although the free-list strategy was claimed to be superior to the multiple-GC strategy by the authors there. In fact, from their simulated data, one cannot discriminate the performance of the two strategies because all discrepant amounts are so small that they fall well within tolerable simulation accuracy. What remains to be compared is the mean search time per allocation attempt; the strategy involving the least search time is the best. The search times for our TC strategy are consistently the lowest, and the free-list strategy the highest, even though an optimistic assumption on the comparison times is assumed for the free-list strategy. This is because the free-list strategy needs a very complicated deallocation process. From the simulated data, it seems that the residence time has only slight impacts on the mean search time for any allocation strategy. Conversely, the requested subcube pattern has considerable influences on the mean search time, especially for the free-list strategy. The last column gives the expected number of right rotating operations per allocation attempt. Clearly, it has very little impact on the time complexity of the TC strategy. Even when it is taken into account, the TC strategy still has lower time complexity than the multiple-GC strategy and much lower than the free-list strategy, since a right rotating operation takes one cycle. The multiple-GC strategy needs, in addition, extra storage for the GC's used.

Simulation runs for H_{10} are also conducted and the results are listed in Table II. In this case, we compare only the free-list and TC strategies, because to generate a proper set of 252 GC's needed for the multiple-GC strategy is not a trivial job (the difficulty in producing an appropriate set of GC's is inherent to the multiple-GC strategy). Again, both strategies lead to almost the same completion time and processor utilization. Also, it can be seen that the TC strategy always has a lower search time, except when the distribution of requested subcube dimensions is normal and the residence time distribution is *uniform*(5,10). When the decreasing subcube distribution is concerned, the mean search time of the free-list strategy is significantly larger. Besides, if the previously mentioned "associative search device" is built, the search time complexity of the TC strategy could be further reduced. For either strategy, the change in residence times has much less impacts on the mean search time than the change in requested subcube patterns.

VI. CONCLUDING REMARKS

In this paper, we have introduced a new hypercube processor allocation strategy. The strategy involves collapsing the binary tree representations of a hypercube successively, dubbed the tree collapsing (TC) strategy. The TC strategy generates its search space dynamically in response to an incoming request, as compared to the multiple-GC strategy [9] where the search space is fixed and predetermined, irrespective of the requested subcube size. This strategy is proved to achieve complete subcube recognition. To facilitate the implementation of the TC strategy, right rotating transforms on the notations of the sets of subcubes (corresponding to the nodes at a certain level of binary tree representations) are suggested.

The TC strategy is compared with the other two allocation strategies capable of accomplishing complete subcube recognition known thus far, namely, the multiple-GC strategy [9] and the free-list strategy [13]. The worst case storage and time complexities are analytically shown to be lower by

TABLE II
SIMULATION RESULTS FOR TASK ALLOCATION IN H_{10}

Residence time distribution	Subcube size distribution	Completion time		Processor utilization		Search time		NR*
		FL*	TC	FL	TC	FL	TC	
uniform (5,10)	uniform	227.1	227.0	58.8	58.9	2999.5	2053.3	39.9
	normal**	133.0	133.4	50.1	50.0	1408.5	2911.7	69.7
	decreasing***	69.2	69.2	35.9	35.9	11238.2	1771.0	24.4
uniform (20,30)	uniform	644.5	645.1	69.7	69.7	2907.3	1870.7	41.3
	normal	327.2	327.0	68.7	68.7	3494.9	2497.3	70.4
	decreasing	141.3	141.2	58.0	58.0	13005.3	1789.0	31.0

*FL: Free-List, NR: number of right rotating operations.
**$p_0 = 0.025$, $p_1 = 0.040$, $p_2 = 0.052$, $p_3 = 0.089$, $p_4 = 0.142$, $p_5 = 0.304$, $p_6 = 0.142$, $p_7 = 0.089$, $p_8 = 0.052$, $p_9 = 0.040$, $p_{10} = 0.025$.
***$p_0 = 0.367$, $p_1 = 0.153$, $p_2 = 0.120$, $p_3 = 0.099$, $p_4 = 0.080$, $p_5 = 0.061$, $p_6 = 0.045$, $p_7 = 0.031$, $p_8 = 0.020$, $p_9 = 0.014$, $p_{10} = 0.010$.

the TC strategy than by the multiple-GC strategy. Extensive simulation runs are also conducted to produce performance measures of interest of the three strategies. From the experimental results, it is observed that all the three strategies give rise to virtually the same completion time (of a sequence of 100 tasks) as well as processor utilization measures. However, the mean search time of the TC strategy is less than those of the other two strategies in most situations.

After repeated allocation and relinquishment of subcubes, a fragmented hypercube happens in which even if there are sufficient hypercube processors available, they do not form a subcube large enough for an incoming task. When a sequence of incoming tasks requests a wide variety of subcube sizes, a fragmented hypercube is likely to arise frequently and would seriously jeopardize its performance level. To deal with this fragmentation problem, a *task migration* approach that can be incorporated into the GC strategy nicely was proposed [11]. It might be interesting to consider task migration or other alternative techniques for resolving fragmentation caused by the TC strategy.

APPENDIX

Before the proof of Theorem 1 is given, three lemmas which characterize properties of collapsed trees generated by Algorithm G are provided next. These lemmas are useful for proving the second part of Theorem 1.

Suppose that S_k and S'_k are subcubes in H_n with addresses $x_{n-1}x_{n-2}\cdots x_1x_0$ and $x'_{n-1}x'_{n-2}\cdots x'_1x'_0$, respectively, and $x_{i_1},\cdots,x_{i_k}$ and $x'_{i'_1},\cdots,x'_{i'_k}$ are *don't care* bits. It is easy to verify that $S_k \neq S'_k$ if $\{i_1,i_2,\cdots,i_k\} \neq \{i'_1,i'_2,\cdots,i'_k\}$ (as they contain different sets of leaf nodes). Therefore, we have the following lemma.

Lemma 2: Given $T_n\langle t_0,t_1,\cdots,t_{n-1}\rangle$ and $T'_n\langle t'_0,t'_1,\cdots,t'_{n-1}\rangle$, T_n and T'_n are overlapped at level $n-k$ if and only if $\{t_i|n-k \leq i \leq n-1\} = \{t'_i|n-k \leq i \leq n-1\}$.

Clearly, there are 2^{n-k} nodes at level $n-k$ of T_n (or T'_n). Consider the two sets $\Upsilon_k = \{S_k|S_k$ is associated with a node at level $n-k$ of $T_n\}$ and $\Upsilon'_k = \{S_k|S_k$ is associated with a node at level $n-k$ of $T'_n\}$. If T_n and T'_n are overlapped at level $n-k$, then $\Upsilon_k = \Upsilon'_k$, since totally there are 2^{n-k} subcubes with dimension k and with the *don't care* bits at the same k directions in their addresses.

Lemma 3: A sequence of C-transforms $\xi_{i_1},\xi_{i_2},\cdots,\xi_{i_j},\cdots,\xi_{i_l}$ operated upon a binary tree representation $T_n(=T_n^{(0)})$ produces collapsed trees $T_n^{(1)},T_n^{(2)},\cdots,T_n^{(j)},\cdots$, and $T_n^{(l)}$, where $0 \leq i_j \leq n-k-1$ for $1 \leq j \leq l$. Then, $T_n^{(0)},T_n^{(1)},T_n^{(2)},\cdots$, and $T_n^{(l)}$ are pairwise nonoverlapped at level $n-k$ if $v-u \leq k$ for any u and v, $0 \leq u < v \leq l$.

Proof of Lemma 3: Assume that the parameter of $T_n^{(x)}$ is denoted by $\langle t_0^{(x)},t_1^{(x)},\cdots,t_{n-1}^{(x)}\rangle$, where $0 \leq x \leq l$. If $v-u \leq k$ for any u and v, $0 \leq u < v \leq l$, according to Lemma 1, we get $t_{n-1}^{(u+1)} = t_{i_{u+1}}^{(u)}$ from applying $\xi_{i_{u+1}}$ to $T_n^{(u)}$, where $t_{i_{u+1}}^{(u)}$ does not belong to $\{t_i^{(u)}|n-k \leq i \leq n-1\}$ since $0 \leq i_j \leq n-k-1$, and $t_{i_{u+1}}^{(u)} \in \{t_i^{(v)}|n-k \leq i \leq n-1\}$ since $t_{i_{u+1}}^{(u)}$ would not be rotated out of the rightmost k positions during the sequence of C-transforms $\xi_{i_{u+2}},\xi_{i_{u+3}},\cdots,\xi_{i_v}$.

From the above statements, we have $\{t_i^{(u)}|n-k \leq i \leq n-1\} \neq \{t_i^{(v)}|n-k \leq i \leq n-1\}$. Then, according to Lemma 2, we arrive at that $T_n^{(u)}$ and $T_n^{(v)}$ are nonoverlapped for any u and v, $0 \leq u < v \leq l$; namely, $T_n^{(0)},T_n^{(1)},T_n^{(2)},\cdots$, and $T_n^{(l)}$ are pairwise nonoverlapped at level $n-k$. □

As an instance, PT_n and the k collapsed trees produced from applying the first row of C-transforms given in Fig. 4 (i.e., k ξ_{n-k-1}'s) to PT_n as well as its subsequent collapsed trees are pairwise nonoverlapped at level $n-k$, since the levels at which C-transforms are carried out are all lower than $n-k$ and the total number of C-transforms performed is k. The same result applies likewise to any other row (i.e., sequence) of C-transforms given in Fig. 4.

Lemma 4: Two distinct sequences of C-transforms $\xi_{i_1},\xi_{i_2},\cdots,\xi_{i_j},\cdots,\xi_{i_l}$ and $\xi_{i'_1},\xi_{i'_2},\cdots,\xi_{i'_j},\cdots,\xi_{i'_l}$ operated upon a binary tree representation T_n produce, respectively, two sets of collapsed trees $\{T_n^{(1)},T_n^{(2)},\cdots,T_n^{(j)},\cdots,T_n^{(l)}\}$ and $\{T_n^{(1)'},T_n^{(2)'},\cdots,T_n^{(j)'},\cdots,T_n^{(l)'}\}$, where $1 \leq l \leq k$ and $0 \leq i_j, i'_j \leq n-k-1$ for $1 \leq j \leq l$, and where for $p \leq r$, $i_p \leq i_r$ and $i'_p \leq i'_r$. Then, any two collapsed trees with one from each set are nonoverlapped at level $n-k$ if $i_1 \neq i'_1$.

Proof of Lemma 4: Assume that the parameters of T_n, $T_n^{(x)}$, and $T_n^{(x)'}$ are denoted, respectively, by $\langle t_0,t_1,\cdots,t_{n-1}\rangle$,

$\langle t_0^{(x)}, t_1^{(x)}, \cdots, t_{n-1}^{(x)}\rangle$, and $\langle t_0^{(x)'}, t_1^{(x)'}, \cdots, t_{n-1}^{(x)'}\rangle$, where $0 \le x \le l$. According to Lemma 1, we get $t_{n-1}^{(1)} = t_{i_1}$ and $t_{n-1}^{(1)'} = t_{i'_1}$ from applying ξ_{i_1} and $\xi_{i'_1}$ to T_n, respectively. During the sequence of C-transforms $\xi_{i_2}, \cdots, \xi_{i_l}$, t_{i_1} would not be rotated out of the rightmost k positions (since $1 \le l \le k$), i.e., $t_{i_1} \in \{t_i^{(x)} | n-k \le i \le n-1\}$ for any x, $1 \le x \le l$. If $i_1 < i'_1$, since $i'_p \le i'_r$ as $p \le r$, t_{i_1} would not rotate into the rightmost position during the sequence of C-transforms $\xi_{i'_2}, \cdots, \xi_{i'_l}$, that is, t_{i_1} does not belong to $\{t_i^{(y)'} | n-k \le i \le n-1\}$ for any y, $1 \le y \le l$.

From the above statements, we get $\{t_i^{(x)} | n-k \le i \le n-1\} \ne \{t_i^{(y)'} | n-k \le i \le n-1\}$ for any x and y, $1 \le x, y \le l$. (The same result can be reached if $i'_1 < i_1$.) According to Lemma 2, the lemma is proved. □

For example, consider the two sets of collapsed trees produced from applying the first two rows of C-transforms shown in Fig. 4 to PT_n. Any collapsed tree in one set and any collapsed tree in the other set are nonoverlapped at level $n-k$, because all the C-transforms in a row (i.e., a sequence) are performed at the same level (which implies that a C-transform at the higher level is not performed ahead of that at the lower level), with the C-transform in one row starting with level $n-k-1$ and in the other row starting with a different level, level $n-k-2$.

With these lemmas, we prove Theorem 1 below.

Proof of Theorem 1: i) Let $[1, 2, \cdots, m]$ denote a sequence of m numbers. Three operators Ω, Ψ, and Δ are defined as follows.

$$[1, 2, \cdots, m]\Omega[1, 2, \cdots, n] = [1, 2, \cdots, m, 1, 2, \cdots, n]$$
$$\Psi(m) = [1, 2, \cdots, m]$$
$$\Psi([1, 2, \cdots, m]) = \Psi(1)\Omega\Psi(2)\Omega\cdots\Omega\Psi(m)$$
$$\Psi^r(x) = \Psi^{r-1}(\Psi(x)) \text{ where } r > 1 \text{ and } x \text{ is a number or a sequence}$$
$$\Delta(\Psi(m)) = \Delta([1, 2, \cdots, m]) = \sum_{i=1}^{m} i.$$

Assume that A_k^i denotes the sum of the numbers in the ith column in Fig. 5, then we have

$$\begin{aligned} A_k^1 &= n-k = C(n-k, 1) \\ A_k^2 &= \Delta(\Psi(n-k)) = 1 + 2 + \cdots + (n-k) \\ &= C(n-k+1, 2) \\ A_k^3 &= \Delta(\Psi^2(n-k)) = \Delta(\Psi(1)\Omega\Psi(2)\Omega\cdots\Omega\Psi(n-k)) \\ &= C(2,2) + C(3,2) + \cdots + C(n-k+1, 2) \\ &= C(n-k+2, 3). \end{aligned}$$

(It is easy to verify by induction that $C(q,q) + C(q+1,q) + \cdots + C(p,q) = C(p+1, q+1)$, where p and q are numbers and $p \ge q$.)

$$\begin{aligned} A_k^4 &= \Delta(\Psi^3(n-k)) = \Delta(\Psi^2(1)\Omega\Psi^2(2)\Omega\cdots\Omega\Psi^2(n-k)) \\ &= C(3,3) + C(4,3) + \ldots + C(n-k+2, 3) \\ &= C(n-k+3, 4) \\ A_k^5 &= \Delta(\Psi^4(n-k)) = \Delta(\Psi^3(1)\Omega\Psi^3(2)\Omega\cdots\Omega\Psi^3(n-k) \\ &= C(4,4) + C(5,4) + \cdots + C(n-k+3, 4) \\ &= C(n-k+4, 5). \end{aligned}$$

From the above, we can derive $A_k^i = C(n-k+i-1, i)$.

Now, given n and k, by induction we can prove that the total number of C-transforms performed in Algorithm G, δ_k, equals $C(n,k) - 1$ as follows.

Base: $\delta_1 = A_1^1 = n - 1 = C(n, 1) - 1$, true

Induction: if $\delta_{l-1} = C(n, l-1) - 1$ holds, that is

$$\begin{aligned} &A_{l-1}^1 + A_{l-1}^2 + A_{l-1}^3 + \cdots + A_{l-1}^{l-1} \\ &= C(n-l+1, 1) + C(n-l+2, 2) + C(n-l+3, 3) \\ &\quad + \cdots + C(n-1, l-1) \\ &= C(n, l-1) - 1, \end{aligned}$$

then δ_l

$$\begin{aligned} &= A_l^1 + A_l^2 + A_l^3 + \cdots + A_l^{l-1} + A_l^l \\ &= C(n-l, 1) + C(n-l+1, 2) + C(n-l+2, 3) \\ &\quad + \cdots + C(n-2, l-1) + C(n-1, l) \\ &= \delta_{l-1} - 1 - (C(n-l+1, 1) + C(n-l+2, 2) \\ &\quad + \cdots + C(n-2, l-2)) + C(n-1, l) \\ &= \delta_{l-1} - 1 - (\delta_{l-1} - C(n-1, l-1)) + C(n-1, l) \\ &= C(n, l) - 1. \end{aligned}$$

ii) Algorithm G has the following facts (as shown in Fig. 4): 1) the highest level to perform C-transforms is $n-k-1$; 2) the maximum length of any sequence of the C-transforms is k; 3) for each sequence, the C-transform is always performed earlier at the lower level than at the higher level; and 4) any two distinct sequences of C-transforms $\xi_{i_1}, \xi_{i_2}, \cdots, \xi_{i_k}$ and $\xi_{i'_1}, \xi_{i'_2}, \cdots, \xi_{i'_k}$ operated upon the primary binary tree representation produce a set of collapsed trees $\{T_n^{(1)}, \cdots, T_n^{(m)}, T_n^{(m+1)}, \cdots, T_n^{(k)}, T_n^{(m+1)'}, \cdots, T_n^{(k)'}\}$, where m is the least integer such that $i_1 = i'_1$, $i_2 = i'_2, \cdots, i_m = i'_m$.

a) From 1), 2), and Lemma 3, it is clear that the collapsed trees produced during any sequence are pairwise nonoverlapped at level $n-k$. b) From 1), 2), 3), 4), and Lemma 4, we reach that any two collapsed trees, respectively, from the two sets $\{T_n^{(m+1)}, \cdots, T_n^{(k)}\}$ and $\{T_n^{(m+1)'}, \cdots, T_n^{(k)'}\}$ are nonoverlapped at level $n-k$ since $i_{m+1} \ne i'_{m+1}$.

According to a) and b) above, the theorem is proved. □

Proof of Theorem 2: Suppose ξ_{i_r} produces $T_n\langle t_0^{(r+1)}, t_1^{(r+1)}, \cdots, t_{n-1}^{(r+1)}\rangle$, where $0 \le i_r \le n-k-1$ and $0 \le r \le k-1$, then if

$$i'_r = i_r + r, \tag{1}$$

we can prove by induction that $\rho_{i'_r}$ produces the notation of $\Phi_{S_k}^{T_n\langle t_0^{(r+1)}, t_1^{(r+1)}, \ldots, t_{n-1}^{(r+1)}\rangle}$ as follows.

Base ($r = 0$): Assume that $a_0 a_1 \cdots a_{n-1}$ is the notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$, where $a_j = *$ if $n-k \le j \le n-1$, otherwise, $a_j = \Box$, and also that ρ_{i_0} produces a notation $a'_0 a'_1 \cdots a'_{n-1}$. According to Definition 5, we can get $a'_j = a_j = \Box$ for $0 \le j < i_0$ from i) of the definition, $a'_{i_0} = a_{n-1} = *$ from iii), and $a'_j = a_{j-1} = *$ for $n-k+1 \le j \le n-1$, or $a'_j = a_{j-1} = \Box$ for $i_0 < j \le n-k$ from ii) since $i_0 \le n-k-1$. We thus prove $a'_0 a'_1 \cdots a'_{n-1}$ to be the notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,i_0-1,i_0+1,\cdots,n-1,i_0\rangle}$ (i.e., $\Phi_{S_k}^{T_n\langle t_0^{(1)},t_1^{(1)},\cdots,t_{n-1}^{(1)}\rangle}$, derived from Lemma 1).

Induction: If $r = m-1, 2 \le m \le k-1$, holds, namely, $\rho_{i'_m}$ operates on the notation of $\Phi_{S_k}^{T_n\langle t_0^{(m)},t_1^{(m)},\cdots,t_{n-1}^{(m)}\rangle}$, say $b_0 b_1 \cdots b_{n-1} = \beta$, where

$$b_u = * \text{ if } u \in \{t_{n-k}^{(m)}, \cdots, t_{n-1}^{(m)}\}, \tag{2}$$

$$b_u = \Box \text{ otherwise,} \tag{3}$$

and produces $b'_0 b'_1 \cdots b'_{n-1} = \beta'$, then, to prove that what $\rho_{i'_m}$ produces is indeed the notation of $\Phi_{S_k}^{T_n\langle t_0^{(m+1)},t_1^{(m+1)},\cdots,t_{n-1}^{(m+1)}\rangle}$ is equivalent to show that if $u \in \{t_{n-k}^{(m+1)}, \cdots, t_{n-1}^{(m+1)}\}$, then $b'_u = *$, otherwise, $b'_u = \Box$, which, from (2) and (3), is to prove that $b'_{t_j^{(m+1)}} = b_{t_j^{(m)}}$ for $0 \le j \le n-1$. Proof of this equation is divided into three steps: to prove $b'_{t_j^{(m+1)}} = b_{t_j^{(m)}}$ for a) $i_m \le j \le n-m-2$, b) $j = n-1$, and c) $0 \le j < i_m$ and $n-m-1 \le j < n-1$, as follows.

a) for $i_m \le j \le n-m-2$

According to Definition 5,

$$b'_v = b_v \qquad \text{for } 0 \le v < i'_m, \tag{4}$$

$$b'_v = b_{v-1} \qquad \text{for } i'_m < v \le n-1, \tag{5}$$

and

$$b'_{i'_m} = b_{n-1}. \tag{6}$$

From Lemma 1, we have

$$t_j^{(m+1)} = t_j^{(m)} \qquad \text{for } 0 \le j < i_m, \tag{7}$$

$$t_j^{(m+1)} = t_{j+1}^{(m)} \qquad \text{for } i_m \le j < n-1, \tag{8}$$

and

$$t_{n-1}^{(m+1)} = t_{i_m}^{(m)}. \tag{9}$$

It is easy to derive

$$t_j^{(x)} = j + x \qquad \text{for } i_{x-1} \le j < n-x \text{ and } 1 \le x \le k. \tag{10}$$

From (10), (1), (5), and (8), since $i_m \ge i_{m-1}$, we get

$$b'_{t_j^{(m+1)}} = b'_{t_{j+1}^{(m)}} = b_{t_j^{(m)}} \qquad \text{for } i_m \le j \le n-m-2.$$

b) for $j = n-1$

The rightmost k don't care bits of the notation of $\Phi_{S_k}^{T_n\langle 0,1,\cdots,n-1\rangle}$ would not get totally rotated out until $\rho_{i'_k}$ is performed, so we have $b_{n-1} = *$. From (6), (1), (10) and (9), we obtain

$$b'_{t_{n-1}^{(m+1)}} = b'_{t_{i_m}^{(m)}} = b'_{i_m+m} = b'_{i'_m} = b_{n-1} = * = b_{t_{n-1}^{(m)}}.$$

c) for $0 \le j < i_m$ and $n-m-1 \le j < n-1$

Equation (4) indicates that the rest bits in β and β' are the same, that is

$$b'_{t_j^{(m+1)}} = b_{t_j^{(m+1)}} \text{ for } 0 \le j < i_m \text{ and } n-m-1 \le j < n-1. \tag{11}$$

Equations (11) and (7) thus give rise to

$$b'_{t_j^{(m+1)}} = b_{t_j^{(m+1)}} = b_{t_j^{(m)}}, \qquad \text{for } 0 \le j < i_m.$$

From (11), (8), and (2), since $n-m-1 \ge n-k > i_m$, we get

$$b'_{t_j^{(m+1)}} = b_{t_j^{(m+1)}} = b_{t_{j+1}^{(m)}} = b_{t_j^{(m)}}, \qquad \text{for } n-m-1 \le j < n-1.$$

Thus, the theorem is proved. $\Box$

ACKNOWLEDGMENT

The authors would like to thank T. Mitra for some initial discussion of this work.

REFERENCES

[1] Y. Saad and M.H. Schultz, "Topological properties of hypercubes," *IEEE Trans. Comput.*, vol. 37, pp. 867–872, July 1988.

[2] C.L. Seitz, "The Cosmic Cube," *Commun. ACM*, vol. 28, no. 1, pp. 22–23, Jan. 1985.

[3] Intel Corp., *A New Direction in Scientific Computing,* Order #28 009-001, Intel Corp., 1985.

[4] W.D. Hillis, *The Connection Machine*. Cambridge, MA: The MIT Press, 1985.

[5] *Ametek System 14 User's Guide: C Edition*, Ametek Computer Research Division, Arcadia, CA, 1986.

[6] J.P. Hayes *et al.*, "A microprocessor-based hypercube supercomputer," *IEEE Micro*, vol. 6, pp. 6–17, Oct. 1986.

[7] H.L. Gustafson, S. Hawkinson, and K. Scott, "The architecture of a homogeneous vector supercomputer," in *Proc. 1986 Int. Conf. Parallel Processing,* Aug. 1986, pp. 649–652.

[8] J.C. Peterson *et al.*, "The Mark III hypercube-ensemble concurrent computer," in *Proc. 1985 Int. Conf. Parallel Processing,* Aug. 1985, pp. 71–73.

[9] M.-S. Chen and K.G. Shin, "Processor allocation in an N-cube multiprocessor using Gray codes," *IEEE Trans. Comput.*, vol. C-36, pp. 1396–1407, Dec. 1987.

[10] K.C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, pp. 623–625, Oct. 1965.

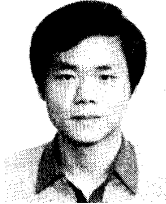
[11] M.-S. Chen and K.G. Shin, "Task migration in hypercube multiprocessors," in *Proc. 16th Annu. Int. Symp. Comput. Architecture,* May 1989, pp. 105–111.

[12] A. Al-Dhelaan and B. Bose, "A new strategy for processor allocation in an N-cube multiprocessor," in *Proc. Phoenix Conf. Comput. and Commun.*, Mar. 1989, pp. 114–118.

[13] J. Kim, C.R. Das, and W. Lin, "A processor allocation scheme for hypercube computers," in *Proc. 1989 Int. Conf. Parallel Processing,* Aug. 1989, pp. II 231–238.

[14] H.S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.

[15] P.-J. Chuang and N.-F. Tzeng, "Dynamic processor allocation in hypercube computers," in *Proc. 17th Annu. Int. Symp. Comput. Architecture,* May 1990, pp. 40–49.



Po-Jen Chuang (S'89) received the B.S. degree from the National Chiao Tung University, Taiwan, Republic of China, in 1978 and the M.S. degree in computer science from the University of Missouri at Columbia in 1988.

Currently, he is a Ph.D. candidate at the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, where he has served as a research/teaching assistant since 1988. His research interests include parallel and distributed processing, fault-tolerant computing, and computer architecture.

Mr. Chuang is a student member of the Association for Computing Machinery.

Nian-Feng Tzeng (S'85–M'86) received the B.S. degree in computer science from National Chiao Tung University, Taiwan, the M.S. degree in electrical engineering from National Taiwan University, Taiwan, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign in 1978, 1980, and 1986, respectively.

He is currently on the faculty of the Center for Advanced Computer Studies at the University of Southwestern Louisiana, Lafayette. From 1986 to 1987, he was a Member of Technical Staff, AT&T Bell Laboratories, Columbus, OH. He was involved in the Cedar supercomputer project at the Center for Supercomputing Research and Development, University of Illinois, Urbana, for more than two years. His current research interest is in the areas of parallel and distributed processing, fault-tolerant computing, and VLSI/WSI-based systems.

Dr. Tzeng is a member of Tau Beta Pi, a member of the Association for Computing Machinery, and the recipient of the Outstanding Paper Award of the 10th International Conference on Distributed Computing Systems, May 1990.