Parallel Algorithms for Hypercube Allocation*

Yeimkuan Chang and *Laxmi N. Bhuyan*
Department of Computer Science, Texas A&M University
College Station, Texas 77843-3112

Abstract – *Parallel algorithms of the hypercube allocation strategies are considered in this paper. Although the sequential algorithms of various hypercube allocation strategies are easier to implement, their worst case time complexities exponentially increase as the dimension of the hypercube increases. We show that the free processors can be utilized to perform the allocation jobs in parallel to improve the efficiency of the hypercube allocation algorithms. A modified parallel algorithm for the single GC strategy is proposed and is shown to be able to recognize more subcubes than the single GC strategy by using the binary reflected Gray code and inverse binary reflected Gray code, without increasing the execution time. Two algorithms for a complete subcube recognition system are also presented and shown to be more efficient and attractive than the sequential one currently used in the hypercube multiprocessor.*

1 Introduction

The hypercube structure has become a widely used architecture in the design of distributed-memory multiprocessor system. Its popularity stems from the compactness of the nodes in the system which results in a logarithmically-growing diameter and degree of the processor nodes. A hypercube with 2^n processors can be topologically represented as an n-dimensional cube in which a processor is located on each one of the 2^n vertices of the cube. Each of the 2^n processors is addressed by a distinct n-bit vector and two processors are connected by a link if and only if their addresses differ in exactly one bit. Subcubes of an n-cube system are denoted by ternary strings in {0, 1, *}, where * is the Don't Care bits which can be replaced by either 0's or 1's. For example, 0*0* is a subcube in a 4-cube system which contains 4 processors with addresses 0, 1, 4, and 5.

Numerous research efforts on the performance evaluation, fault tolerance, and embeddability of hypercubes [1, 2] have been reported. Several commercial hypercube multiprocessors have been built, such as Intel iPSC [3], and nCUBE[4]. When a task or an application program arrives at a hypercube multiprocessor, the required number of processors is assigned to the task by the host processor. Upon the completion of the task, the processors used by the task are released or deallocated. The processor allocation involves two steps. The first step is to determine the number of processors that should be allocated for executing an incoming task. It is assumed that a complete subcube is required by the incoming task. Otherwise, a subcube of dimension d is allocated, where $2^{d-1} < p \leq 2^d$ and p is the number of requested processors. The second step in the processor allocation is to locate an appropriate subcube and assign it to the incoming task such that the system utilization is maximized and the system fragmentation is minimized. As a result, the delay for an incoming task to be scheduled is minimized. In short, a good allocation scheme keeps the dimension of the free subcubes as large as possible while the processors are allocated and deallocated. The current hypercube multiprocessors allocate tasks based on a simple buddy strategy, as explained later [3, 4].

A few subcube allocation policies have been proposed in the literature. These policies concentrate on developing a method or a data structure that helps the allocation procedure to quickly find a first-fit available subcube of the requested size and assign it to the request. The performance metrics used in comparisons of different allocation algorithms are usually the number of recognizable subcubes and allocation time efficiency. The number of recognizable subcubes is closely related to the processor utilizations. One category of the allocation policies, called bit-mapping schemes, uses tree structures to facilitate the processor allocation. Buddy, gray code(GC) and multiple-GC [5], modified buddy [6], and tree collapsing(TC) [7] strategies belong to this category. Another category of the allocation policies, called list schemes, maintains $n+1$ lists in which the i^{th} list contains the available i-dimensional subcubes. The elements in the lists are mutually disjoint. Free list strategy[8] and maximal set of subcubes(MSS)[9] are in this category. There are also many other variations of the allocation policies that use graphs to quickly find out the available subcubes. The examples of them are the MSS-based algorithm using consensus graphs in [9] and the prime cube-based algorithm using prime cube(PC) graphs in [10]. Another approach is the weight allocation strategy(WAS) [11] which uses the weights of the processors to select the best subcube in order to reduce the fragmentation of the processors. The weight of a processor is defined as the number of its neighbors which are busy.

In the sequential approach, the host processor is responsible for all the computations of the availability of subcubes. Since the time complexities of the proposed approaches are high, these papers [5, 7, 8]

*This research has been partly supported by NSF grant MIP-9002353

briefly discuss about the parallel implementation of their strategies. One parallel approach commonly employed in [5, 7, 8] is to distribute the computations to some predetermined processors. The results from these predetermined processors are then collected by the host processor to make the final decision. In [5], the number of predetermined processors used by the multiple-GC strategy for a complete subcube recognition is $C^n_{\lfloor \frac{n}{2} \rfloor}$. The TC strategy in [7] uses C^n_d processors to achieve the complete subcube recognition when a subcube of size d is requested. The free list strategy employs $n+1$ processors, each of which maintains the list of one dimension. Two drawbacks of these parallel algorithms are as follows. First, if the processors are initially selected and dedicated solely to executing the allocation program, then the number of processors available for executing the incoming tasks is reduced, leading to degraded system performance. The second drawback is that the worse case time complexity is still of the order of 2^n which exponentially increases as the dimension of the system increases.

Another parallel approach used in [12] utilizes the free processors to perform the allocation jobs. Only tree type allocation strategies were considered. We adopt a similar approach but parallelize all the existing hypercube allocation strategies. Also the worst case time complexity of our best parallel algorithm is $\sum_{i=1}^{d} \lceil \frac{C^n_i}{2^i} \rceil$ which is much better than $C^n_d \times d$ obtained in [12]. We intend to distribute the computations executed on the host processor in the sequential approach to the free processors by keeping the free processors busy and responsible all the time. This approach keeps the system more utilized, improves the execution time of the allocation and reduces the waiting time of the incoming tasks.

In this paper, we first parallelize the buddy, single GC, multiple-GC, and TC strategies. A modified parallel algorithm for the single GC strategy is proposed and is shown to be able to recognize more subcubes than the single GC strategy by using the binary reflected Gray code and inverse binary reflected Gray code, without increasing the execution time. Two algorithms of a complete subcube recognition system are also presented and shown to be more efficient and attractive than the sequential one currently used in the hypercube multiprocessor. The rest of the paper is organized as follows. The parallel algorithms of the buddy and GC strategies are given in Section 2 and 3, respectively. Efficient algorithms of the complete subcube recognition system are presented in section 4. A selection algorithm suited for the best-fit strategy such as WAS is given in section 5. Finally, concluding remarks are presented in Section 6.

2 The Buddy Strategy

The buddy strategy is originated from the memory allocation scheme. This is the simplest among all the strategies and is applied to commercial multiprocessors, such as nCUBE[4]. It has been shown that the buddy strategy is statically optimal if the release of processors is not considered. In other words, the buddy strategy fails to grant a request only if there is no sufficient number of available processors to satisfy the request. However, the buddy strategy is no longer optimal if the release of the processors is considered when a request is completed.

This strategy is easily explained by a binary tree in which the leaf nodes are labeled as 0 to 2^n-1 from the left to the right. The labels of the leaf nodes represent the addresses of processors in the system. In the sequential algorithm, when a d-cube is requested the system checks the availability of the processors corresponding to the leaf nodes of the subtrees rooted at the $(n-d)^{th}$ level of the binary tree. Formally, the system checks the availability of 2^d processors whose addresses range from $2^d \times i$ to $2^d \times (i+1) - 1$, where $i = 0$ to $2^{n-d} - 1$. It can be easily observed that the number of d-cubes recognized by the buddy strategy is 2^{n-d}.

Parallel Algorithm of the Buddy Strategy

To parallelize the buddy strategy, we use the the same divide-and-conquer technique usually used in the tree structure. Each available processor with address $a_{n-1}..a_0$ has local variables $RESULT$ and ACK and executes the following algorithm when a d-cube is requested.

```
1  ACK = 0, RESULT = 1
2  for i = 0 to d − 1 in parallel do
3      if a_i is 0 and processor a_{n-1}..a_{i+1}10..0
         is available then
4          receive ACK
5      else ACK = 0
6      if ACK is 0 then RESULT = 0
7      if a_i is 1 and processor a_{n-1}..a_{i+1}00..0
         is available then
8          send RESULT to processor a_{n-1}..a_{i+1}00..0
9  end for
10 if RESULT is 1 then send a_{n-1}..a_d**...* to
     the host processor
```

Having developed the above algorithm, we propose below a way by which the parallel algorithm can be implemented. The executable image of the subprogram of the parallel algorithm must be preloaded at each processor's memory. As a task arrives, the associated information such as the requested subcube size d is loaded to each processor, from the host processor. This loading process can be done in a constant time since the host processor is connected to every processor in the system as implemented in $nCUBE$ multiprocessor [4]. Then the parallel algorithm is executed.

At the end of algorithm, the processors $a_{n-1}..a_d00..0$ with a value 1 in variable ACK discovers a free d-cube $a_{n-1}..a_d$**...* and sends the address of the d-cube to the host processor. The host processor then selects the first d-cube it receives, assigns the d-cube to the incoming task and disregards the other d-cubes arriving at the host processor. Obviously, this algorithm takes d time units, each of which is the time processors take for sending and receiving results from other processors. The sequential buddy strategy takes $O(2^n)$ time units for the worst case.

Notice that the loading process described here are applicable to all the parallel algorithms given in this

paper. The subcube selection method adopted by the host processor is only applicable to the first-fit approaches such as the buddy and GC strategies. For the best-fit approach a different selection method is needed as described later for the WAS.

Extension of the buddy strategy

The tree collapsing(TC) strategy [7] is an extension of the buddy strategy that has a complete subcube recognition ability. The TC strategy tries to find a subcube according to the buddy strategy. If unsuccessful, it will generate another binary tree with different labeling which can be generated by the collapsing tree generation method developed in [7]. Basically, each binary tree contains 2^{n-d} d-cubes each of which has d *'s at d fixed bit positions and 0's or 1's at the other $n-d$ bit positions in its ternary representation. For example, 0*0*, 0*1*, 1*0*, and 1*1* are the 2-cubes recognized by the TC strategy based on the binary tree labeled with (0,1,4,5,2,3,6,7,8,9,12,13,10,11,14,15). Hence, at most C_d^n binary trees are needed to allocate a d-cube. As stated in the introduction, the distributed algorithm developed for TC strategy in [7] uses C_d^n processors, each of which computes the availability of subcubes recognized by its corresponding binary tree. Each involved processor sends the result back to the host processor, indicating whether a subcube of the requested size is available. The worst case time complexity is $O(2^n)$.

To make the TC strategy more efficient, we parallelize the TC strategy by using the parallel algorithm of the buddy strategy C_d^n times with C_d^n binary trees. Thus the time complexity of the parallel algorithm of the TC strategy becomes $O(d \times C_d^n)$ which is polynomial and more efficient than the distributed algorithm developed in [7].

3 The Gray Code Strategy

The single GC strategy [5] is similar to the buddy strategy except for the labeling of the leaf nodes. The leaf nodes are labeled by a sequence of binary numbers where any two consecutive numbers have only one different bit out of the n bits, based on the binary reflected Gray Code(BRGC). Let g_n denote the binary reflected Gray Code mapped from {0, .., 2^n-1} to n-bit binary strings and $G_n = \{g_n(0), .., g_n(2^n-1)\}$. The subscript n will be omitted later if there is no ambiguity. The leaf nodes are labeled as $g_n(0)$ to $g_n(2^n-1)$ from the left to the right. G_n is obtained by the following recursive expression with parameters $\{p_1, p_2, .. p_n\}$,

$$G_1 = \{0,1\}$$
$$G_n = \{G_{k-1}^{0/r_k}, (G_{k-1}^*)^{1/r_k}\}, 2 \le k \le n. \quad (1)$$

Here r_k is the partial rank of p_k in $\{p_1, p_2, .. p_k\}$. G_{k-1}^{b/r_k} is the set of k-bit binary strings which are constructed by inserting a bit with value (0 or 1) into the position between $r_k{}^{th}$ bit and $(r_k-1)^{th}$ bit of the elements in G_{k-1}, assuming there exist two dummy bits, $r_k{}^{th}$ and 0^{th} bits. G_{k-1}^* is the sequence of binary strings obtained by reversing the order of the strings in G_{k-1}. The standard GC is the G_n with parameters {1, 2, .., n}. An example of the standard BRGC G_4 of a 4-cube system is shown in Fig.1.

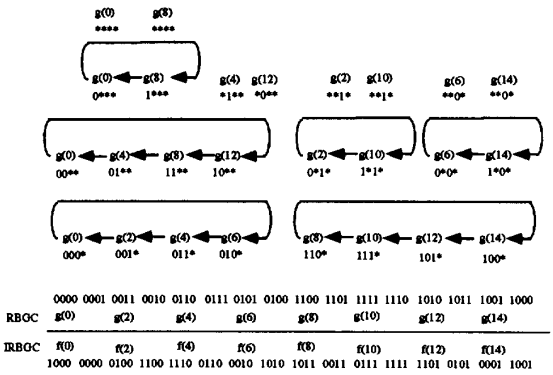

Figure 1: The allocation process of the modified single GC strategy in a 4-cube system.

The sequential allocation algorithm of the single GC strategy for allocating a d-cube is to check the availability of 2^d processors rooted at two consecutive nodes of the $(n-d)^{th}$ level of the GC binary tree. Formally, the system checks the availability of 2^d processors whose addresses range from $2^{d-1} \times i$ to $(2^{d-1} \times (i+2) - 1) \bmod 2^{d-1}$, where $i = 0$ to $2^{n-d-1} - 1$. It can also be easily observed that the number of the recognizable subcubes is double compared to the buddy strategy. The time complexity of the single GC strategy, however is $O(2^n)$. Our parallel version of the single GC strategy spends exactly the same time, d time units, as the buddy strategy to compute the availability of every subcube recognized by the single GC strategy. The algorithm is illustrated as follows.

Parallel Algorithm of the Single GC Strategy

G = $\{g(0), g(1), .., g(2^n-1)\}$ Each available processor P_α with address $\alpha = a_{n-1}..a_0$ has local variables $RESULT$ and ACK and executes the following algorithm when a d-cube is requested.

```
1  ACK = 0, RESULT = 1
2  for i = 0 to d − 1 in parallel do
3      k = g((g^{-1}(α) + 2^i) mod 2^n)
4      l = g((g^{-1}(α) − 2^i) mod 2^n)
5      if P_l is available then
6          send RESULT to P_l
7      if P_k is available then
8          receive ACK
9      else ACK = 0
10     if ACK is 0 then RESULT = 0
11 End for
12 if RESULT is 1 then send the found d-cube
   described in the text to the host processor
```

At the end of the parallel single GC algorithm, the

processor $g^{-1}(\alpha) = b_{n-1}..b_{d-1}00..0$ with a value 1 in variable $RESULT$ discovers a free d-cube of 2^d processors whose addresses range from $g(2^{d-1} \times i)$ to $g((2^{d-1} \times (i+2) - 1) \bmod 2^{d-1})$, where $i = b_{n-1}..b_{d-1}$.

The Modified Single GC Strategy

While developing the parallel algorithms for the buddy and single GC strategies, we discover that not all the processors are responsible for the processor allocation all the time. For example, after the first iteration of the algorithm, at most one half of the processors labeled with $g(i)$, where i is odd, will become idle since they do not send results to or receive results from other processors. Therefore we develop a modified algorithm for the single GC strategy to be able to recognize more subcubes than the single GC strategy by keeping all the available processors busy and responsible for the processor allocation all the time. In addition to the binary reflected Gray code(BRGC) $G_n = \{g(0), g(1), .., g(2^n - 1)\}$, we also map the processors with the inverse binary reflected Gray code(IBRGC) $F_n = \{f(0), f(1), .., f(2^n - 1)\}$. IBRGC F_n listed below is obtained from a similar recursive expression as BRGC with an inverse parameters set $\{p_n, .., p_1\}$.

$$F_1 = \{1, 0\}$$
$$F_n = \{F_{k-1}^{0/r_k}, (F_{k-1}^{*})^{1/r_k}\},\ 2 \le k \le n, \quad (2)$$

where the definitions of r_k and F^* are the same as BRGC. An example of the standard IBRGC F_4 of a 4-cube system is shown in Fig.1.

Notice that if a processor α has an even value of $g^{-1}(\alpha)$ then it will have an odd value of $f^{-1}(\alpha)$ and vice versa. In the first step of the parallel algorithm of the modified single GC strategy, processors $g(x)$ and $f(y)$ send results to processors $g(x-1)$ and $f(y-1)$, respectively, if x and y are odd. Processor $g(x)$ and $f(y)$ expect to receive results from processors $g(x+1)$ and $f(y+1)$, respectively if x and y are even. Thus we can easily divide the processors into two sets of processors, $\{g(0), g(2), .., g(2^{n-1})\}$ and $\{f(0), f(2), .., f(2^{n-1})\}$. After the first step of the algorithm, these two sets of processors perform the processor allocation independently. In the following iterations, each processor sends result to its child processor and receives result from its parent processor in a similar way.

The relationship of the processors is shown in Fig.2 in which four processors are related and the child processors are pointed to by directed arrows from their parent processors and vice versa. The variable D of a free processor, shown in Fig.1, is the dimension index which is the index of the leftmost Don't Care bit in the ternary representation of the subcube plus one and is used to compute the addresses of its parent and the child processors. To be more clear, Fig.1 shows that processors $g(0)$, $g(2)$, $g(4)$, and $g(6)$ are related to each other since D is 1 at the end of the first iteration of the parallel algorithm. Processor $g(0)$ is the parent of the processor $g(6)$ and therefore will send a result to the processor $g(6)$. Processor $g(0)$ is the child of the processor $g(2)$ and will expect to receive a result from processor $g(2)$.

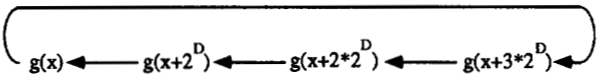

Figure 2: The child and parent relationship of the processors.

0000*	*0000	0****	****0
000**	**000	*0***	***0*
00*0*	*0*00	**0**	
00***	***00	***0*	
0*0**	**0*0		
0**0*	*0**0		
*0*0*			

Figure 3: The subcubes of a 5-cube system recognized by the modified single GC strategy.

Fig.3 shows the subcubes that can be recognized by the parallel algorithm of the modified single GC strategy in a 5-cube system. For the processors labeled with RBGC, we can observe that the subcubes recognized by the parallel algorithm of the modified single GC strategy are constructed from the 1-cubes each of whose ternary representations has a Don't-Care bit at the 0^{th} bit position. The 2-cubes recognized by the algorithm are constructed by assigning the 1^{st} bit with * and also assigning 3^{rd} bit with *. In general, the d-cubes are based on the $(d-1)$-cubes and are constructed by assigning the bit next to the leftmost Don't Care bit of the ternary representation of the (d-1)-cubes to * and also assigning the bit which is two bits away from the leftmost Don't Care bit to *.

The parallel algorithm is shown in the Appendix. The variable $Cube$ in the algorithm shown in the Appendix stores the subcube discovered at the current iteration. The variable D is an index of the leftmost(rightmost) Don't Care bit in variable $Cube$ plus 1 for BRGC(IBRGC). At the end of the algorithm, the processor with value 1 in variable $RESULT$ discovers a free subcube stored in variable $Cube$. It can be easily observed that the modified single GC strategy takes exactly the same time, d time units, as the buddy and the single GC strategy, but recognizes more subcubes. Table 1 shows the number of subcubes that an n-cube system has and the number of the subcubes that can be recognized by the parallel algorithms of the complete recognition approaches, modified GC strategy, and single GC strategy.

Multiple GC strategy

In the multiple-GC strategy, $C^n_{\lfloor \frac{n}{2} \rfloor}$ gray codes are used for a complete subcube recognition [5]. For each incoming task with requested size d, the system follows

Dim	CR	MS GC	S GC
n	1	1	1
$n-1$	$2n$	$2n$	4
$n-2$	$2(n^2-n)$	$2(n^2-3n)$	8
3	$C_3^n \times 2^{n-3}$	2^n	2^{n-2}
2	$C_2^n \times 2^{n-2}$	2^n	2^{n-1}
1	$n \times 2^{n-1}$	2^n	2^n
0	2^n	2^n	2^n

Table 1: Numbers of the subcubes recognized by a complete recognition(CR) system, the single GC (S GC)strategy and the modified single GC (MS GC)strategy.

the allocation procedure of the single GC strategy to find the first available d-cube based on the binary tree with one gray code. If there is no free d-cube in the tree, then the second binary tree is generated with another gray code and the search is continued until an available d-cube is found or all the $C^n_{\lfloor \frac{n}{2} \rfloor}$ gray codes are exploited. Therefore, we can have a parallel algorithm for the multiple GC strategy with a time complexity of $O(d \times C^n_{\lfloor \frac{n}{2} \rfloor})$ by using the same technique as the parallel algorithm of the TC strategy. Notice that although the number of the subcubes recognized by the single GC strategy is double that of the buddy strategy, the time complexity of the multiple GC strategy is worse than the TC strategy. The reason is that the subcubes checked in one GC may be redundantly checked in another GC of the multiple GC strategy.

4 Complete Subcube Recognition

In order to achieve a complete subcube recognition, a processor allocation algorithm must have the ability to check all the possible subcubes in the system. There are $C_d^n \times 2^{n-d}$ d-cubes in an n-cube system. Thus a sequential algorithm with the complete subcube recognition ability proceeds by checking the availability of each of the 2^n processors in each of the $C_d^n \times 2^{n-d}$ d-cubes, which gives us a $O(C_d^n \times 2^n)$ time complexity. In order to improve the efficiency of the allocation algorithm, we can have a parallel algorithm with time complexity $O(C_d^n \times d)$ that achieves a complete subcube recognition by employing the buddy strategy C_d^n times with C_d^n different buddy trees, i.e. the parallel algorithm of the TC strategy.

In the following we present two parallel algorithms that achieve a complete subcube recognition and are more efficient than the parallel TC algorithm. The basic idea is to uniformly distribute the computations that are supposed to be executed in the host processor in the sequential algorithm to all the free processors. The algorithms involve two phases. The first phase, the subcube assignment phase, assigns the subcubes to their corresponding leader processors. The leader of a subcube is one of the processors in the subcube determined by the subcube assignment scheme. As we know, there are $C_d^n \times 2^{n-d}$ d-cubes in an n-cube system, the subcube assignment scheme can distribute the $C_d^n \times 2^{n-d}$ d-cubes uniformly to the 2^n processors in the system. Thus the number of the d-cubes processed in a processor is less than or equal to $\lceil \frac{C_d^n \times 2^{n-d}}{2^n} \rceil = \lceil \frac{C_d^n}{2^d} \rceil$. Notice that, according to the definition of the leader of a subcube, if the leader is busy then the subcube must not be available. Thus only the subcubes that are computed by the free processors need to be considered in the processor allocation procedure. The subcube assignment scheme is given as follows.

1. Initialize the variable counter[α] to 0 for each processor α.
2. For a processor with address $\alpha = (a_{n-1}, \ldots, a_0)$, generate C_d^n d-cubes, $(a_{n-1}, \ldots, x_{d-1}, \ldots, x_0, \ldots, a_0)$ that contain processor α, where x_i are DON'T CARE symbols, and $0 \leq i < d$. Sort these C_d^n d-cubes by their index of x_i.
3. For the k^{th} d-cube in the sorted sequence of the C_d^n d-cubes, where k is from 0 to $C_d^n - 1$ and $y = k \ mod \ 2^d = (y_{d-1}, \ldots, y_0)$, x_i with y_i, $0 \leq i \leq d-1$, and calculate $\beta = (a_{n-1}, \ldots, y_{d-1}, \ldots, y_0, \ldots, a_0)$.
4. If the counter[β] is not larger than $\lceil \frac{C_d^n}{2^d} \rceil$, then assign the d-cube, $(a_{n-1}, \ldots, x_{d-1}, \ldots, x_0, \ldots, a_0)$, to the processor β. Compute the complementary d-cube by complementing all the bits of $(a_{n-1}, \ldots, x_{d-1}, \ldots, x_0, \ldots, a_0)$ except DON'T CARE bits, and assign this complementary d-cube to processor $\overline{a} = (\overline{a_{n-1}}, \overline{a_{n-2}}, \ldots, \overline{a_1}, \overline{a_0})$.
5. else, set $k = k + 1$, and $y = k \ mod \ 2^d = (y_{d-1}, \ldots, y_0)$ and calculate $(a_{n-1}, \ldots, y_{d-1}, \ldots, y_0, \ldots, a_0)$.
6. Goto step 3 until all the $C_d^n \times 2^{n-d}$ d-cubes are assigned to a processor.

Notice that the subcube assignments according to the above scheme can be determined off-line and incorporated in the processor allocation algorithm for each free processor. The entire program consists of the host and free processor subprograms. Whenever a subcube is requested, the host processor decides the cube size, i.e. d, and loads the free processors with d. The subprogram in each free processor then follows the same procedure as the sequential algorithm and finds a free d-cube. After the subprogram is finished on each free processor, the result is sent to the host processor. Then the host processor picks the first result(a free subcube) it receives and allocates it to the incoming task, and disregards the other results arriving at the host processor. Therefore, this gives us a time complexity of $O(\lceil \frac{C_d^n}{2^d} \rceil \times 2^d) \simeq O(C_d^n)$ which is better than the parallel TC algorithm developed in section 2.

Since the above algorithm involves a lot of redundant computations among the free processors, we can

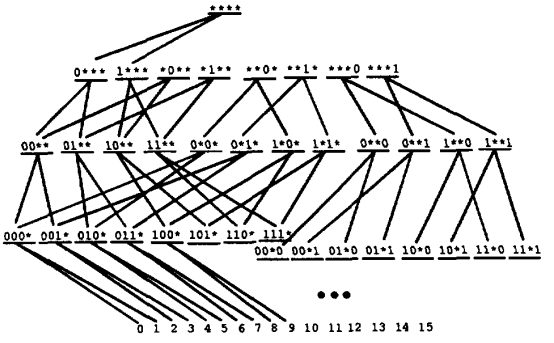

Figure 4: An example of modified single GC strategy in a 4-cube system.

use the divide-and-conquer technique to further improve the efficiency. This can be accomplished by using the partial results of lower dimension subcubes to compute the availability of the higher dimension subcubes. The first phase is the same as the above algorithm. In the second phase, we use the subcube tree to facilitate the development of the parallel algorithm. Fig.4 shows the subcube tree of a 4-cube system for allocating a 3-cube. Each k-cube ($1 \leq k \leq d$) that need to be computed is decomposed into two $(k-1)$-cubes at the position of the leftmost Don't Care bit of its ternary representation. For example, the 3-cube 0*** is decomposed into two 2-cubes, 00** and 01** in the subcube tree shown in Fig.4. From the decomposition procedure, $C_{d-k}^{n-k} \times 2^{n-d+k}$ $(d-k)$-cubes need to be computed, where $1 \leq k \leq d$. Each k-cube receives two results of $(k-1)$-cubes from its child processors in the subcube tree. Since a leader processor is responsible for at most $\lceil \frac{C_i^n}{2^i} \rceil$ d-cubes, the time complexity of the parallel algorithm is easily obtained to be $2 \times \sum_{i=1}^{d} \lceil \frac{C_i^n}{2^i} \rceil = O(\sum_{i=1}^{d} \lceil \frac{C_i^n}{2^i} \rceil)$.

In the above complexity analysis, we assume that the time for passing results between processors is independent of the distance between processors. It is always true in the modern hypercube multiprocessors such as nCUBE because of the wormhole routing technique [4]. We also ignore the message contention in the system since the messages passed for the allocation jobs are small and are less likely to interfere with the messages passed in the subcubes that have already been allocated.

5 The Best-Fit Strategies

As we should see, the main tasks of the parallel hypercubes allocation algorithms developed in previous sections are as follows. Each free processor searches for an available subcube of the requested size among some predetermined subcubes and sends the found subcube to the host processor. Then the host processor selects the first subcube it receives and assigns the subcube to the incoming task. Basically, these algorithms are first-fit approaches. In this section, we extend the technique used in previous sections to the best fit approach, the weight allocation strategy(WAS) [11]. Each free processor finds the best subcube among the predetermined $\lceil \frac{C_d^n}{2^d} \rceil$ d-cubes using WAS. Then the global best d-cube is selected by the selection procedure which will be described later. We do not attempt to parallelize the other best fit approaches such as the list type strategies, the MSS-based [9] and free list [8] strategies since the list structure is not easy to parallelize. In the following, we first briefly describe the WAS and then develop an efficient method to select the best cube for allocation.

Weight Allocation Strategy

WAS is a hypercube allocation strategy developed in [11] that is based on the weight sums of the subcubes in the system. The weight of a free processor is the number of its neighbors that are already allocated to tasks. The weight sum of a free subcube is defined as the sum of the weights of the processors in the subcube. WAS selects a free subcube whose weight sum is maximal among all the available subcubes of the requested size. The sequential algorithm of the WAS, executed on the host processor, is given as follows.

1. Set $d = |I_j|$, where $|I_j|$ is the dimension of the subcube requested by an incoming task, I_j.
2. Compute the weight of each free processor.
3. Determine the availability of a d-cube by using its corresponding track bits.
4. If there are available d-cubes, compute the weight sums of all the available d-cubes.
   (a) If there is an available d-cube whose weight sum is $(n-d) \times 2^d$ then select it and quit.
   (b) Find the d-cube whose weight sum is maximum among the available d-cubes.
      i. If weight sums of two d-cubes, A and B, are equal, compute weight vectors, $\vec{a} = (a_n, ..., a_0)$ and $\vec{b} = (b_n, ..., b_0)$ for A and B, where a_i and b_i are number of processors whose weight is i, and $0 \leq i \leq n$. If $\vec{a} > \vec{b}$, select B. If $\vec{a} < \vec{b}$, select A.
      ii. If $\vec{a} = \vec{b}$, compute the sum w_a of weights of free processors after temporarily setting the corresponding track bits of A to ones. Similarly compute w_b. If $w_a < w_b$, then select A. Else, if $w_a > w_b$ select B.
      iii. If $w_a = w_b$, compute a weight cardinality vector $(c_n, ..., c_0)$ after temporarily setting the corresponding track bits of A to ones, where c_i is the number of free processors whose weights are i. Similarly, compute the weight cardinality vector $(d_n, ..., d_0)$ for B. If $(d_n, ..., d_0) < (c_n, ..., c_0)$ then select B, otherwise select A.

5. Set the track bits corresponding to the selected d-cube in step 3 to 1's and assign the d-cube to I_j.

6. If there is no available d-cube, put the task I_j to waiting list until a subcube is released.

Step 4 is the major part of the allocation algorithm. It first computes the weight sums of all the available d-cubes. Then it selects the d-cube which has the maximum value of weight sum among all the available d-cubes. Since it is possible that there are more than one d-cubes with the same weight sum which is maximum, further comparisons are performed as stated in step 4(b). However, according to the computer simulation in [11], these cases are rare. The step 4(a) is a special case of the algorithm. It finds a d-cube with weight sum $(n-d) \times 2^d$ which is maximal weight sum in any situation. In other words, no other d-cube can have weight sum greater than $(n-d) \times 2^d$. Thus it is not necessary to continue the algorithm and the algorithm can quit at that point.

To compute the availability of a d-cube in the WAS, $O(n \times 2^d)$ time units are needed for the worst case. There are at most $C_d^n \times 2^{n-d}$ available d-cubes in the system. Thus the time complexity of the sequential algorithm of the WAS is $O(nC_d^n \times 2^n)$. By using the subcube assignment scheme, each free processor handles $\lceil \frac{C_d^n}{2^d} \rceil$ d-cubes and executes the sequential algorithm as the host processor. Notice that if we omit the steps 4(b)ii and 4(b)iii which do not occur very often, then we can use the subcube tree to further improve the efficiency as we did before.

The final step is to collect the results from the free processors. We cannot follow the same approach as the first-fit algorithm where each free processor sends the result to the host processor. If the number of free processors is large, the host processor becomes a bottleneck since it needs to receive all the results from the free processors before it selects a d-cube. We develop the following parallel algorithm for selecting the best subcube while avoiding the bottleneck at the host processor.

1. If there exists a free d-cube whose weight sum is $2^d \times (n-d)$, then inform the host that the best d-cube is found, and then quit.

2. Repeat the following from $i = 1$ to n,
   for processor $j = 0$ to 2^n - 1 **in parallel do**

   (a) If processor j is free and a variable DONE is 0, choose a processor h, where h has a lowest address among 2^i processors, DONE bit of processor is 0, and $\lfloor \frac{j}{2^i} \rfloor \times 2^i \leq h < \lfloor \frac{j+2^i}{2^i} \rfloor \times 2^i$.

   (b) Send the result (a found d-cube) to processor h.

   (c) set the DONE to 1 if $j \neq h$.

3. Send the result of the only free processor with DONE 0, to the host.

	sequential	parallel
buddy	$O(n)$[13, 14]	$O(d)$
single GC	$O(2^n)$	$O(d)$
multiple GC	$O(C^n_{\lfloor \frac{n}{2} \rfloor} \times 2^n)$	$O(C^n_{\lfloor \frac{n}{2} \rfloor} \times d)$
TC	$O(C_d^n \times 2^n)$	$O(C_d^n \times d)$
free list	$O(n \times 2^n)$	$O(2^n)$
complete 1	$O(C_d^n \times 2^n)$	$O(C_d^n)$
complete 2	$O(C_d^n \times 2^n)$	$O(\sum_{i=1}^{d} \lceil \frac{C_i^n}{2^i} \rceil)$
WAS	$O(n \times C_d^n \times 2^n)$	$O(n \times C_d^n)$

Table 2: The worst case time complexities of the sequential and parallel algorithms for various hypercube allocation strategies, where complete 1 and 2 are the first and second parallel algorithms respectively developed in the section of the complete subcube recognition.

Note that, in step 2 of the above parallel algorithm, only 2 processors out of 2^i processors in an i-cube have value 0 of DONE bit, and one of the higher address processors sends the result to the other processor. The communication for passing results in one i-cube is independent of the communications in another i-cube. Therefore, the total time complexity of the parallel algorithm of WAS becomes $O(n\lceil \frac{C_d^n}{2^d} \rceil \times 2^d)$.

6 Concluding Remarks

In this paper, we develop parallel algorithms for the tree type of hypercube allocation strategies, such as buddy, single GC, multiple GC, and TC strategies. We also propose a modified single GC parallel algorithm that recognizes more subcubes than the single GC strategy with the same execution time. Two efficient parallel algorithms are developed for the complete subcube recognition by utilizing all the free processors. Finally, we extend the parallel algorithms to the best-fit strategy WAS and achieve the same order of the complexity as the tree type first fit approaches. A summary of the complexities of the various parallel algorithms is given in Table 2. We can see that the complexities of the parallel algorithms are dramatically decreased. These results make the best fit approach such as WAS more attractive since the best fit approach reduces the fragmentation of the processors in the system.

References

[1] L. Bhuyan and D. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Networks," *IEEE Transactions on Computers*, pp. 323–333, Apr 1984.

[2] D. A. Reed and D. C. Grunwald, "The Performance of Multicomputer Interconnection Networks," *IEEE Computer*, pp. 63–73, June 1987.

[3] J. Rattner, "Concurrent Processing: A New Direction in Scientific Computing," *AFIPS Conference Proc.*, pp. 157–166, vol.54 1985.

[4] nCUBE Corporation, *nCUBE 2 Processor Manual*, nCUBE Corporation, Dec. 1990.

[5] M. S. Chen and K. G. Shin, "Processor Allocation in an N-Cube Multiprocessor Using Gray Codes," *IEEE Transactions on Computers*, pp. 1396–1407, Dec. 1987.

[6] A. Al-Dhelaan and B. Bose, "A New Strategy for Processors Allocation in an N-Cube Multiprocessor," *Proc. Int'l Pheonix conference On Computers communications*, pp. 114–118, 1989.

[7] P. J. Chuang and N. F. Tzeng, "Dynamic processor Allocation in hypercube Computers," *Proc. Int'l Conference on Computer Architectures*, pp. 40–49, 1990.

[8] J. Kim, C. R. Das, and W. Lin, "A Top-down processor Allocation scheme for hypercube Computers," *IEEE Transactions on Parallel and Distributed Systems*, pp. 20–30, January 1991.

[9] S. Dutt and J. P. Hayes, "Subcube Allocation in Hypercube Computers," *IEEE Transactions on Computers*, pp. 341–351, March 1991.

[10] H. Wang and Q. Yang, "Prime Cube Graph Approach for Processor Allocation in Hypercube Multiprocessors," *Proc. Int'l Conference on Parallel Processing*, pp. I–25–32, 1991.

[11] Y. Chang and L. N. Bhuyan, "A New Approach to Hypercube Allocation using Weights," Technical Report 93-004, Computer Science Department, Texas A&M University, 1993.

[12] M. Livingston and Q. F. Stout, "Parallel Allocation Algorithms for Hypercubes and Meshes," *Fourth Hypercube Concurrent Computers and Applications*, pp. 59–66, 1989.

[13] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley Publishing Company, 1978.

[14] D. D. Sharma and D. Pradhan, "A Novel Approach for Subcube Allocation in Hypercube Multiprocessors," In *International Symposium on Parallel and Distributed Processing*, pp. 336–345, Dec. 1992.

7 Appendix

Parallel Algorithm of the Modified Single GC Strategy

Standard binary reflected Gray Code: $G_n = \{g(0), g(1), .., g(2^n - 1)\}$ Inverse standard binary reflected Gray Code: $F_n = \{f(0), f(1), .., f(2^n - 1)\}$ Each available processor P_α with address $\alpha = a_{n-1}..a_0$ has local variables $RESULT$, A, D, and $Cube$. The following algorithm is executed when a d-cube is requested.

```
1  Cube = α, A = 1, and D = 1
2  x = g^{-1}(α) and y = f^{-1}(α)
3  For i = 0 to d − 1 in parallel do
4   If D ≥ n then quit
5   If i is 0 then
6    If x mod 2 is 1 then
7     send RESULT to P_{g(x−1)}
8    If x mod 2 is 0 then
9     If P_{g(x+1)} is available then
10     receive A
11     Cube[0] = '*'
12    else A = 0
13    if A is 0 then RESULT = 0
14   If y mod 2 is 1 then
15    send A to P_{f(y−1)}
16   If x mod 2 is 0 then
17    If P_{f(y+1)} is available then
18     receive A
19     Cube[n − 1] = '*'
20    else A = 0
21    if A is 0 then RESULT = 0
22  else
23   if x is even then
24    If D < n − 1 then
25     If x − 2^D < 0 then
26      u = g(x + 3 × 2^D), v = g(x + 2^D), and D = D + 1
27     else if x − 2 × 2^D < 0 then
28      u = g(x − 2^D), v = g(x + 2^D), and D = D + 2
29     else if x − 3 × 2^D < 0 then
30      u = g(x − 2^D), v = g(x + 2^D), and D = D + 1
31     else if x − 4 × 2^D < 0 then
32      u = g(x − 2^D), v = g(x − 3 × 2^D), and D = D + 2
33    else
34     If x − 2^D < 0 then
35      u = g(x + 2^D)
36     else u = g(x − 2^D)
37     v = u and D = D + 1
38    send RESULT to P_u
39    if P_v is available then
40     receive A
41     Cube[D − 1] = '*'
42    else A = 0
43    if A = 0 then RESULT = 0
44   if y is even then
45    If y − 2^D < 0 then
46     u = f(y + 3 × 2^D), v = f(y + 2^D), and D = D + 1
47    else if y − 2 × 2^D < 0 then
48     u = f(y − 2^D), v = f(y + 2^D), and D = D + 2
49    else if y − 3 × 2^D < 0 then
50     u = f(y − 2^D), v = f(y + 2^D), and D = D + 1
51    else if y − 4 × 2^D < 0 then
52     u = f(y − 2^D), v = f(y − 3 × 2^D), and D = D + 2
53    send RESULT to P_u
54    if P_v is available then
55     receive A
56     Cube[n − 1 − D − 1] = '*'
57    else A = 0
58 end for
59 if RESULT is 1 then sent the found d-cube stored
       in variable Cube to the host processor
```