Fall 10-31-1994

Processor allocation for partitionable multiprocessor systems

Nicholaos C. Antoniou
New Jersey Institute of Technology

ABSTRACT

PROCESSOR ALLOCATION FOR PARTITIONABLE MULTIPROCESSOR SYSTEMS

by
Nicholaos C. Antoniou

The processor allocation problem in an n-dimensional hypercube multiprocessor is similar to the conventional memory allocation problem. The main objective is to maximize the utilization of available resources as well as minimize the inherent system fragmentation. In this thesis, a new processor allocation strategy is proposed, and compared with the existing strategies, such as, the *Buddy strategy*, the *Single Gray Code strategy* (SGC), the *Multiple Gray Code* (MGC), and the *Maximal Set of Subcubes* (MSS). We will show that our proposed processor allocation strategy outperforms the existing strategies, by having the advantage of being able to allocate unused processors to other jobs/algorithms.

PROCESSOR ALLOCATION FOR PARTITIONABLE MULTIPROCESSOR SYSTEMS

by
Nicholaos C. Antoniou

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

October 1994

APPROVAL PAGE

PROCESSOR ALLOCATION FOR PARTITIONABLE MULTIPROCESSOR SYSTEMS

Nicholaos C. Antoniou

Dr. Edwin Hou, Thesis Advisor — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Sotirios Ziavras, Committee Member — / / — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Kyriakos Mouskos, Committee Member — Date
Assistant Professor of Civil and Enviromental Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Nicholaos C. Antoniou

Degree: Master of Science in Electrical Engineering

Date: October 1994

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1994
- Bachelor of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1992

Major: Electrical Engineering

This work is dedicated to
the memory of my late father
Christodoulos Antoniou
and my beloved mother
Ioanna Antoniou

ACKNOWLEDGMENT

The author wishes to express his sincere gratitude to his advisor, Professor Edwin Hou, for his guidance, friendship, understanding, and moral support throughout this research.

A very special thanks to Professors Sotirios Ziavras and Kyriakos Mouskos for serving as members of the committee and for providing valuable comments and suggestions for this thesis.

The author greatly acknowledges the assistance of Professors Gregory Kriegsmann and Michael Porter of the Department of Mathematics in the form of a graduate assistantship.

An additional word of thanks to Professor Michael Porter for being so supportive and understanding to the author as his supervisor in the computer laboratory of the Department of Mathematics and for making his high-tech computer equipment available to him from which this thesis has greatly benefited.

An additional word of thanks to Professors Gregory Kriegsmann and Demitrius Papageorgiou of the Department of Mathematics for the financial assistance in the form of a graduate assistantship for the Summer of 1994.

The author also wishes to express appreciation to his brother Dr. Antonios Antoniou and his best friend Michael Sideras for their moral support, timely help and suggestions. Also, the author wishes to express a special thanks to all his friends and relatives for their support during his studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Hypercube multiprocessors have been drawing considerable attention mainly due to their structural regularity for easy construction and high potential for the parallel execution of various algorithms. A hypercube multiprocessor is often viewed as a *personal* supercomputer since it has the potential to surpass the fastest supercomputers at a fraction of the cost, Denning [7]. Numerous research efforts related to hypercube architectures, operating systems, programming languages, etc., have been under taken, and several research and commercial hypercube multiprocessors have been built.

A task arriving at a hypercube multiprocessor, called an *incoming request*, can be specified in a graphic form and must be assigned *optimally* to a subcube in the multiprocessor for execution. Upon completion of the execution, the subcube used for the task must be released for later use. Efficient allocation and/or deallocation of node processors in a hypercube multiprocessor is a key to its performance and utilization. The processor allocation in a hypercube multiprocessor consists of two steps: 1) *determination* of the size of a subcube to accommodate an incomming task, and 2) *location* of a subcube of the size determined by 1) within the hypercube multiprocessor. The first step is treated in Chen and Shin [9] and the second step is the subject of this thesis. Results on the existence of subcubes of certain dimensions after link/node failures have been reported elsewhere, Becker and Simon [8].

This thesis addresses the problem of locating available subcubes after a sequence of subcube allocations and relinquishments, thereby distinguishing this work from those described in Becker and Simon [8], and Chen and Shin [9].

Given a node addressing scheme, a set of *contiguous* nodes forms a subcube in an n-cube multiprocessor, similar to a set of memory pages forming a memory segment. This fact implies a close resemblance of the processor allocation problem in the n-cube multiprocessor to the conventional memory allocation problem. In both problems, we want to maximize the utilization of available resources and also minimize inherent systems fragmentation. Five allocation strategies for the n-cube multiprocessor are addressed here: the *buddy strategy* which is based on the buddy system, Knowlton [16], the *single gray code (SGC) strategy* which is based on the *binary-reflected gray code*(BRGC), Chen and Shin [6], the *multiple gray code (MGC) strategy* which is also based on the BRGC, Chen and Shin [6], the *maximal set of subcubes (MSS) strategy* which is based on the notion of a maximal subset of subcubes, Dutt and Hayes [1], and our newly proposed strategy, *allocation strategy I (ASI)*.

We will explore the properties of the *buddy*, *SGC*, *MSG*, and *MSS* allocation strategies, and our processor allocation strategy *allocation strategy I* will be proposed to remedy the processor underutilization problem of the above mentioned allocation strategies. The performances of the current strategies and the proposed one will be comparatively analyzed.

The thesis is organized as follows. Chapter 2 introduces the necessary definitions and notations as well as a review of the four current allocation strategies. Chapter 3 provides the formal description of the problem as well as a detailed account of the proposed algorithm and the approach. Chapter 4 presents simulation results of the various allocation strategies implemented and a discussion of the performance of these allocation strategies. The thesis concludes with chapter 5, which also includes future work and references.

CHAPTER 2

DEFINITIONS, NOTATIONS AND REVIEW OF EXISTING ALLOCATION ALGORITHMS

This chapter introduces the necessary definitions and notations used in this thesis as well as a brief study of existing representative processor allocation strategies. For simplicity sake, we will ignore the overhead for allocation and deallocation, and thus jobs are not penalized for these overheads. This simplification gives a strong benefit of the doubt to the more sophisticated strategies, which in practice incur considerably more overhead that the simpler strategies do. Even with this rather unfair advantage, however, we show that the most sophisticated strategies are generally capable of little or no improvement in performance over the simplest strategies.

2.1 Definitions and Notations

For a formal description of the n-cube structure, it is necessary to define the product of graphs as follows.

Definition 2.1 *Let $G_p = (V_p, E_p)$ be the product of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, denoted by $G_p = G_1 \times G_2$. Then, $V_p = V_1 \times V_2$ and two nodes $u = (u_1, u_2)$ and $v = (v_1, v_2)$ are adjacent in G_p iff [$u_1 = v_1$ and u_2 adjacent to v_2] or [u_1 adjacent to v_1 and $u_2 = v_2$].*

An n-cube can now be defined as follows.

Definition 2.2 *An n-cube, Q_n, is defined recursively as follows.*

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

b) $Q_n = K_2 \times Q_{n-1}$, where K_2 is the complete graph with two nodes.

Let $\sum$ be the ternary symbol set $\{0, 1, *\}$, $*$ is a DON'T CARE symbol. Then, every subcube of an n-cube can be uniquely represented by a string of symbols in $\sum$. Such a string of ternary symbols is called the *address* of the corresponding subcube. For example, the address of the subcube Q_2 in a Q_4 which consists of nodes 0010, 0011, 0110, and 0111 is $0*1*$. Note that the number of $*$'s in the address of a subcube is the dimension or size of the subcube. For convenience, the rightmost coordinate of an address of a subcube in the n-cube will be referred to as *direction 1* and the second to the rightmost coordinate as *direction 2*, and so on. Let $*^k$ denote k consecutive $*$'s. A ternary string will also be used to denote the set of integers in binary representation that result from setting each $*$ to 0 or 1. For example, $01**$ means the set of integers, $\{4, 5, 6, 7\}$.

Let $\{g_1, g_2, \cdots, g_n\}$ be a sequence of distinct integers. For $1 \leq i \leq n$ the *partial rank* r_i of g_i is defined as the rank of g_i in the set $\{g_1, g_2, \cdots, g_i\}$ when the set is rearranged in ascending order. For example, when $\{g_1, g_2, g_3\} = \{3, 1, 2\}$, $r_1 = 3$, $r_2 = 1$, $r_3 = 2$. Let A be a sequence of binary strings of length $n-1$, $n > 1$. Then, a sequence of binary strings of length n, denoted by $A^{b\backslash k}$, $b \in \{0, 1\}$, can be obtained by either inserting a bit b into the position immediately right of the kth bit of every string in A if $1 \leq k \leq n-1$, or prefixing a bit b to every string in A if $k = n$. Also, let A^* denote the sequence of binary strings obtained from A by reversing the order of the strings in A. For example, if $A = \{00, 01, 11, 10\}$, we have $A^{1\backslash 2} = \{010, 011, 111, 110\}$, $A^{1\backslash 3} = \{100, 101, 111, 110\}$, and $A^* = \{10, 11, 01, 00\}$. Using the above notation, Gray codes are defined formally as follows.

Definition 2.3 *Let G_n be the GC with parameters g_i, $1 \leq i \leq n$, where $\{g_1, g_2, \cdots, g_n\}$ is a permutation of $Z_n \equiv \{1, 2, \cdots, n\}$. Then, G_n is defined recursively as follows.*

$$G_1 = \{0, 1\},$$
$$G_k = \{G_{k-1}^{0\backslash r_k}, (G_{k-1}^*)^{1\backslash r_k}\},\ 2 \leq k \leq n,$$

000	000
001	010
011	110
010	100
110	101
111	111
101	011
100	001
(a)	(b)

Figure 2.1 Illustration of Gray codes. (a) BRGC. (b)A GC with $\{g_1, g_2, g_3\} = \{2, 3, 1\}$.

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

For example, for a given GC with parameters $\{g_1, g_2, g_3\} = \{2, 3, 1\}$, we get $\{r_1, r_2, r_3\} = \{2, 3, 1\}$, $G_1 = \{0, 1\}$, $G_2 = \{\underline{0}0, \underline{0}1, \underline{1}1, \underline{0}1\}$, $G_3 = \{00\underline{0}, 01\underline{0}, 11\underline{0}, 10\underline{0}, 00\underline{1}, 01\underline{1}, 11\underline{1}, 10\underline{1}\}$, where the newly inserted bits are underlined. It is worth mentioning that the above definition is a generalization of the Gray codes commonly encountered in the literature, Chan and Saad [11], and that the *binary reflected Gray code*(BRGC), the most frequently used GC, can be obtained readily from this definition by letting $g_i = i$, $1 \leq i \leq n$. Figure 2.1 (a) and (b) shows, respectively, a BRGC and a GC with $\{g_1, g_2, g_3\} = \{2, 3, 1\}$. Note that a GC with parameters g_i, $i = 1, \cdots, n$, can be obtained be permuting the BRGC in such a way that the direction i of the BRGC becomes the direction g_i of this GC. For simplicity, unless specified otherwise, G_n will henceforth be referred to as the BRGC.

A set of contiguous integers is called a *region* and let $\#[a, b] = \{k \mid a \leq k \leq b, k \in I^+\}$, where I^+ denotes the set of positive integers. Let $B_n(m)$ denote the binary representation of an integer m with n-bits and $G_n(m)$ be the BRGC representation of m. Also, the notation $\lfloor b \rfloor$ is used to denote the largest integer which is less than or equal to b, and $\lceil b \rceil$ denotes the smallest integer which is greater than or equal to b. Let $|S|$ denote the cardinality of the set S.

2.2 Processor Allocation Strategies

This section briefly describes and explores the properties of existing representative processor allocation strategies.

2.2.1 The Buddy Strategy

The Buddy strategy, originally proposed for storage allocation by Knowlton [16] in 1965, has since been applied to processor allocation in hypercubes. For a job requesting a subcube of dimension k within a hypercube of dimension n, the Buddy strategy is as follows: Find the smallest integer j, $0 =< j =< 2^{n-k} - 1$, such that all processors in the subcube $\#[j2^k, (j+1)2^k - 1]$ are available, and allocate these processors to the job. If no such j exists, no subcube can currently be allocated to the job. Although quite simple, Buddy has poor subcube recognition abilities relative to other strategies. For this study, we use a particularly efficient implementation of Buddy that uses $n+1$ doubly linked lists (one for each subcube dimension), together with an array of pointers (one for each possible subcube) to keep track of available subcubes. Under this algorithm, the worst-case time complexity of both allocation and deallocation is $O(n)$, where n is the dimension of a hypercube containing 2^n processors.

One form of the buddy strategy was investigated in Purdom and Stigler [14] and also implemented in the NCUBE/six multiprocessor, NCUBE Corp. [12]. Since there are 2^n processor nodes in a Q_n, 2^n allocation bits are used to keep track of the availability of all the nodes. An allocation bit with value 0 (1) indicates the availability (unavailability) of the corresponding node. The buddy strategy consists of two parts, processor *allocation* and processor *relinquishment*, which are outlined below.

Processor Allocation:

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

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

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

Processor Relinquishment:

Step 1. Reset every pth allocation bit to 0, where $B_n(p) \in q$ and q is the address of a released subcube.

This strategy can be explained by the binary tree in Figure 2.2. The level where the root node resides is numbered 0, and the nodes in level i are associated with subcubes of dimension $n - i$. A node in this binary tree is available only if all of its offsprings are available. When an incoming request needs a Q_k, the buddy strategy searches the level $n - k$ of the tree from left to right and allocates the first available subcube to the request. The processors associated with allocation bits in $\#[m2^k,\ (m+1)2^k - 1]$ always constitute a Q_k whose address is $B_{n-k}(m)^{*k}$.

Similarly to the conventional memory allocation, whenever a processor allocation or relinquishment takes place, the subcube to be allocated or released must be associated with a region of contiguous allocation bits. Static allocation is

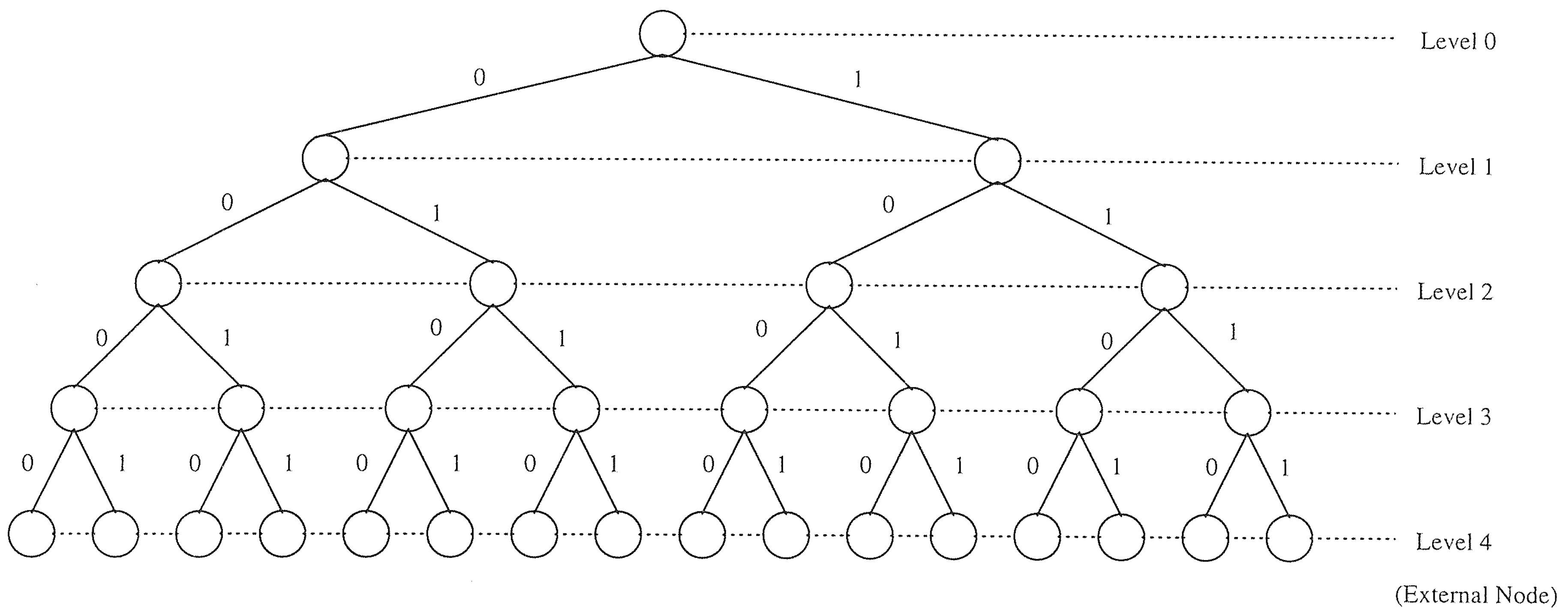


Figure 2.2 The complete binary tree for the allocation strategy using the buddy system.

concerned only with how to accommodate incoming requests without considering processor relinquishment. Figure 2.3 shows a simple example of static allocation. It is easy to observe that Q_4 can accommodate the incoming request sequence $\{I_1, \cdots, I_8\}$ even if the order of the requests in the sequence was arbitrarily shuffled. As we shall prove later, this is not a coincidence, but rather, a result of the *static optimality*. An allocation strategy is said to be *statically optimal* if a Q_n using the strategy can accommodate any input request sequence $\{I_k\}_{i=1}^{k}$ iff $\sum_{i=1}^{k} 2^{|I_i|} \leq 2^n$, where $|I_j|$ is the dimension of a subcube required to accommodate the request I_j. We shall prove that below, in Theorem 2.1, that the buddy strategy is statistically optimal.

To facilitate the proof of static optimality of the buddy strategy, it is necessary to introduce the following definition and two lemmas.

Definition 2.4 *A region $b_n b_{n-1} \cdots b_{n-k+1} {}^{*n-k}$ is said to be a hole if this region is available, but $b_n b_{n-1} \cdots \bar{b}_{n-k+1} {}^{*}$ is not, where $b_i \in \{0, 1\}$, $i = n, n-1, \cdots, n-k+1$.*

Note that a region is unavailable if any one of the allocation bits associated with the region is set to 1. Clearly, an allocation bit reset to 0 must always belong to one and only one hole. Let $\{h_i(j)\}_{i=1}^{u}$ denote the sequence of holes which result from allocating subcubes to the request sequence $\{I_1, I_2, \cdots, I_j\}$, where $u$is the number of holes. Order the hole sequence in such a way that $h_p(j)$must lie before $h_q(j)$iff $p < q$. Then we have the following two useful lemmas.

Lemma 2.1 *Let $\{h_i(j)\}_{i=1}^{u}$ be the hole sequence in a Q_n following the allocation of subcubes to the request sequence $\{I_r\}_{r=1}^{j}$. If $h_i(j) = b_n b_{n-1} \cdots b_{n-k+1} {}^{*n-k}$ for some i, then $b_{n-k+1} = 1$.*

Lemma 2.2 $\forall$ *j and $j \leq u$, $\sum_{i=1}^{k-1} 2^{|h_i(j)|} < 2^{|h_k(j)|}$, where $\{h_i(j)\}_{i=1}^{u}$is a hole sequence and $|h_m(j)|$ is the number of $*$'s in the address of $H_m(j)$.*

$I_1 = Q_0$ $\qquad$ $I_5 = Q_1$

$I_2 = Q_2$ $\qquad$ $I_6 = Q_2$

$I_3 = Q_0$ $\qquad$ $I_7 = Q_0$

$I_4 = Q_0$ $\qquad$ $I_8 = Q_1$

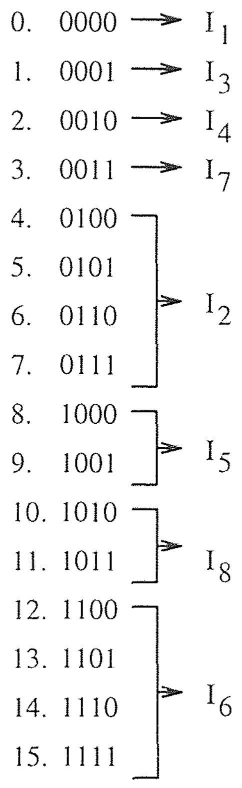


Figure 2.3 Allocation strategy using the buddy system

These lemmas lead to the following important result.

Theorem 2.1 *The buddy strategy is statistically optimal.*

Note that when a Q_k is needed, the buddy strategy searches for a region of allocation bits with 0's whose addresses start with an integral multiple of 2^k. This in turn implies that there are only $2^{n-k} Q_k$'s within the n-cube multiprocessor recognizable by the buddy strategy. Consequently, the buddy strategy underutilizes processors in the n-cube multiprocessor.

2.2.2 Single Gray Code (SGC)

The Single Gray Code (SGC) allocation strategy, proposed by Chen and Shin [6] in 1987, is based on the *binary-reflected gray code* (BRGC), which is the gray code having parameters $1, 2, ..., n$. A BRGC is a sequence of $2^n n$-bit codes, G_n, and is defined recursively as follows: For $G_1 = (0,1)$and $G_n = (x_0, x_1, ..., x_{2^n-1})$, $G_{n+1} = (0x_0, 0x_1, ..., 0x_{2^n-2}, 0x_{2^n-1}, 1x_{2^n-1}, 1x_{2^n-2}, ..., 1x_1, 1x_0)$. For example, the BRGC $G_3 = \{000, 001, 011, 010, 110, 111, 101, 100\}$. Let $G_n(m)$denote the mth code in the sequence G_n. The availability of processors is represented by using 2^nallocation bits. A bit having value 0 indicates that the corresponding processor is available, whereas 1 indicates a processor in use. For a job requesting a subcube of dimension k, we find the smallest integer $j$such that all of the $(m \bmod 2^n)$th allocation bits are 0's, where $m \in \#[j2^{k-1}, (j+2)2^{k-1} - 1]$, and allocate the corresponding free subcube be setting the bits to 1. The number of subcubes recognizable by SGC is twice that of buddy strategy. The allocation time complexity of this algorithm is $O(2^n)$, and for deallocation is $O(2^k)$.

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

Processor Allocation:

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

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

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

Processor Relinquishment:

Step 1. Reset every pth allocation bit to 0, where $G_n(p) \in q$ and q is the address of a subcube released.

Since the nodes corresponding to the first and last allocation bits are adjacent to each other, circular search is allowed in the GC strategy. To show that the nodes associated with those allocation bits in $\#[m2^k, (m+2)2^{k-1} - 1]$ constitute a Q_k, consider another procedure for generating the BRGC, Chan and Saad [11]. As mentioned before, one can assume, without loss of generality, that the GC strategy uses the BRGC only. It is proved in Reingolg, Nievergelt and Deo [13], that this procedure indeed generates the BRGC. Given a k-bit BRGC $G_k = \{d_0, d_1, \cdots, d_{2^k-1}\}$, a $(k+1)$ bit BRGC can be generated by $G_{k+1} = \{d_0 0, d_0 1, d_1 1, d_1 0, d_2 0, d_2 1, \cdots, d_{2^k-1} 1, d_{2^k-1} 0\}$.

This procedure can be described by a complete binary tree as in Figure 2.4. The address of every external node is determined by the coded bits in the path from the root to the external node, and the BRGC is then obtained by the addresses of external nodes left to right.

Similarly to the binary tree in Figure 2.2, the nodes in level $n-k$ are associated with Q_k's. It is easy to see from the scheme of coding edges of the tree that two

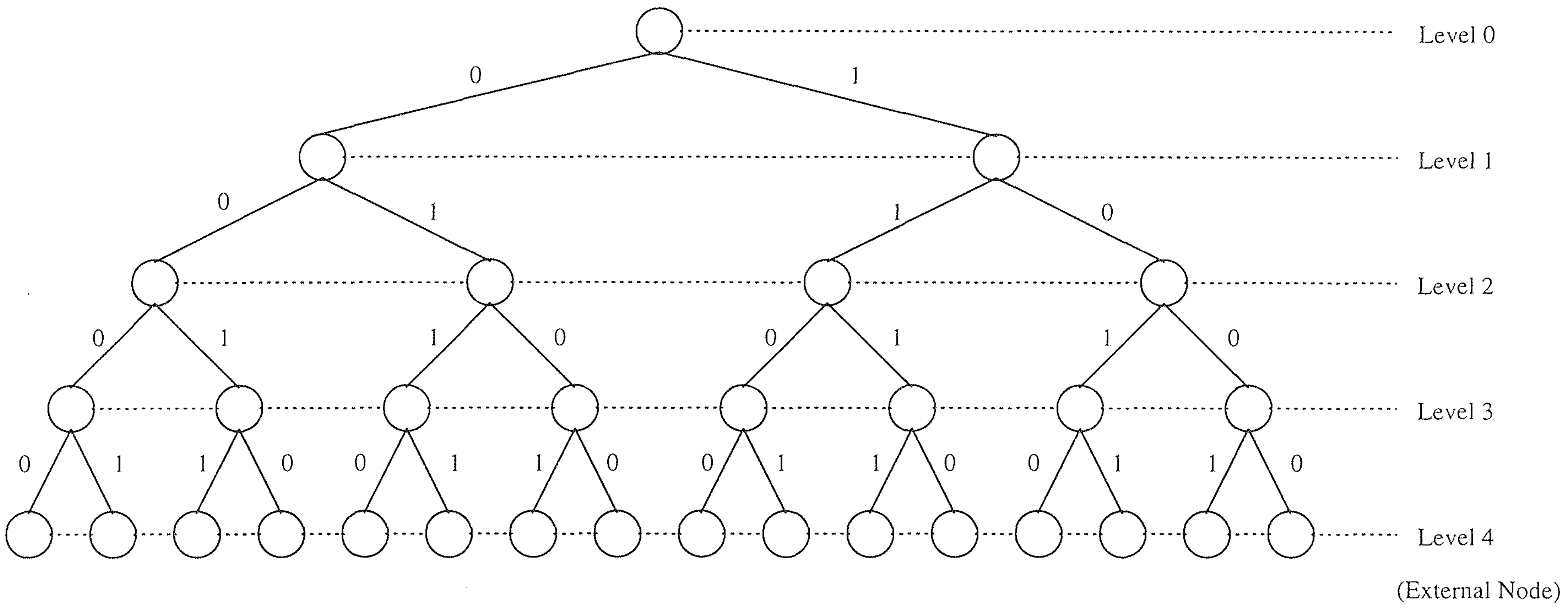


Figure 2.4 The complete binary tree for the allocation atrategy using the binary reflected Gray code.

Table 2.1 The Number of Subcubes Recognizable by the Buddy System and a SGC

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

adjacent nodes in the $(n-k+1)$th level form a Q_k even when they don't have the same immediate predecessor. Therefore, when a Q_k is requested, the GC strategy searches from left to right for two adjacent available nodes in level $n-k+1$, rather than searching for an available node in level $n-k$. A Q_k will thus be searched for in the regions whose addresses start with an integral multiple of 2^{k-1} instead of 2^k. Recall that the later was used by the buddy strategy. This means that the number of subcubes recognizable by a GC is twice that by the buddy strategy. The number of subcubes recognizable by each of the two strategies is presented in Table 2.1.

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

Theorem 2.2 *The GC strategy is statistically optimal.*

An example of the GC strategy is given in Figure 2.5, where the input request sequence is the same as that in Figure 2.3. It can be observed that the GC strategy outperforms the buddy strategy in the first-fit search and will *pack* incoming requests more densely, thus making larger contiguous regions available than the buddy strategy can.

2.2.3 Multiple Gray Code (MGC)

Although the Single Gray Code strategy has better subcube recognition than the Buddy strategy does, SGC cannot generally identify all subcubes of a given dimension. Because different gray codes are associated with different sets of recognizable subcubes, subcube recognition can be improved by using more than one gray codes. It is therefore important to investigate the relationship between the number of GC's employed and the corresponding subcube recognition ability. A new gray code with parameters $\{g_1, g_2, ..., g_n\}$ is obtained by permuting the BRGC in such a way that the ith bit (numbered from right to left, beginning with 1) of each member of the BRGC becomes the g_ith bit of that member in the new gray code. So, G_3 with parameters {2, 3, 1} is {000, 010, 110, 100, 101, 111, 011, 001}. Out of the $n!$ distinct gray codes that exist for an n-cube, Chen and Shin [6] have shown that the minimal number required for complete subcube recognition is $\binom{n}{n/2}$. For example, 20 gray codes are required for n=6; 252 for n=10; 3432 for n=14 and 184,756 for n=20. The time complexity of allocation is $O((\binom{n}{\lfloor n/2 \rfloor})2^n)$, whereas deallocation is $O(2^k)$.

2.2.3.1 Subcube Recognition Ability of a SGC

Let $\{g_1, g_2, ..., g_n\}$ be a permutation of Z_n. By permuting the ith direction of the BRGC to the g_ith direction, one can obtain a GC with parameters g_i, $i = 1, \cdots, n$. Since there are $n!$ permutations of n distinct numbers, there are $n!$ distinct GC's for the n-cube multiprocessor.

$I_1 = Q_0$ $\qquad$ $I_5 = Q_1$

$I_2 = Q_2$ $\qquad$ $I_6 = Q_2$

$I_3 = Q_0$ $\qquad$ $I_7 = Q_0$

$I_4 = Q_0$ $\qquad$ $I_8 = Q_1$

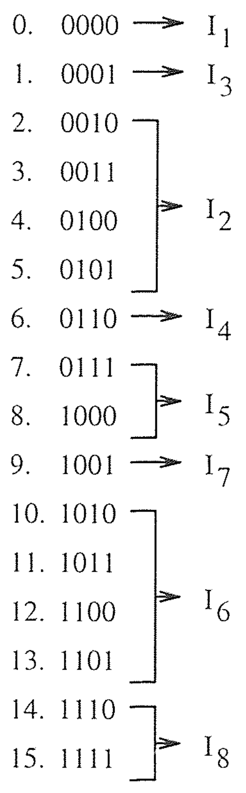


Figure 2.5 Allocation strategy using the binary reflected Gray code.

Moreover, the subcube recognition ability of each GC can be determined by the following theorem.

Theorem 2.3 *A subcube Q_k with the address $b_n b_{n-1} \cdots b_1$ can be recognized by a GC with parameters g_1, $1 \leq i \leq n$, iff any of the following three conditions is satisfied.*

a) $b_{g_i} = *$, $1 \leq i \leq k$.

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

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

An illustrative example of the subcube recognition ability of a 4-bit BRGC is shown in Figure 2.6. It is interesting to observe that the node addresses of a subcube recognizable by a GC are contiguous in that GC. This is the very reason that a GC can be used to detect the availability of subcubes with the linear search designed for the conventional memory allocation.

Note that Theorem 2.3 provides a necessary and sufficient condition for the availability of a subcube to be recognized by a GC. Since different GC's are associated with different sets of recognizable subcubes, processor improves as the number of GC's used in an allocation strategy increases. Consider an allocation strategy which uses three GC's with the following parameters: $\{g_j^3\}_{j=1}^{5} = \{1,2,3,4,5\}$, $\{g_j^2\}_{j=1}^{5} = \{2,5,1,3,4\}$, and $\{g_j^3\}_{j=1}^{5} = \{3,1,4,5,2\}$. Then, the set of subcubes recognizable by this allocation strategy can be determined by Theorem 2.3 and shown in Figure 2.7 with the trivial cases for Q_0 and Q_5 omitted. Also, it is shown from Theorem 2.3 that every subcube must be recognizable by at least on GC and that the complete subcube recognition can be achieved if all the $n!$ GC's are used. However, we naturally want to reduce, if possible, the number of GC's required for complete subcube recognition in order to minimize the search overhead associated with MGC's. More on this will be discussed in the following subsubsection.

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

Figure 2.6 Illustration of Theorem 2.3 when $\{g_1, g_2, g_3, g_4\} = \{1, 2, 3, 4\}$ when $d \in \{0, 1\}$.

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

Figure 2.7 Recognizable subcubes by the given Gray codes, $\{g_1, g_2, g_3, g_4, g_5\} = \{1,2,3,4,5\}, \{2,5,1,3,4\}, \{3,1,4,5,2\}$ where $d \in \{0,1\}$.

2.2.3.2 The Number of GC's for Complete Subcube Recognition Let S be a set of strings which are permutations of Z_n. S is said to have the C *property*, Chen and Shin [6], if for any k distinct numbers from Z_n there is at least one string $s \in S$ such that these k numbers are the first k numbers of s.

Lemma 2.3 *Let M_k be the set of all combinations of k distinct integers out of Z_n, $0 \leq k \leq n$, and let $x \leq y$ denote that all the integers in a combination x are contained in another combination y. Then*

a) there is a one-to-one function $f : M_i \rightarrow M_{i+1}$, $0 \leq i \leq \lfloor n/2 \rfloor - 1$, such that $\forall x \in M_i, x \leq f(x)$, and

b) there is a one-to-one function $g : M_{i+1} \rightarrow M_i$, $\lfloor n/2 \rfloor \leq i \leq n-1$, such that $g(x) \leq x$.

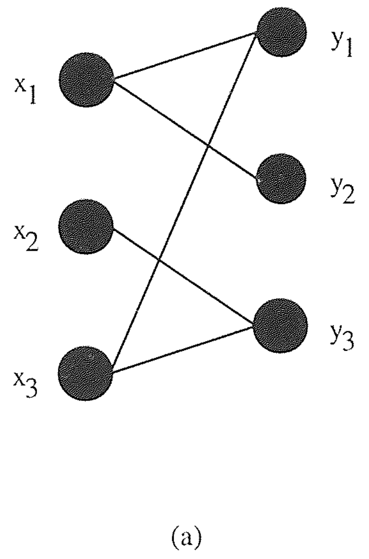
It is necessary to introduce the Theorem of Matching, Liu [15], to prove Lemma 2.3.

Theorem 2.4 *Theorem of Matching [15]. In a bipartite graph $G = (V, E)$, a complete matching from $X \subseteq V$ to $Y \subseteq V$ exists iff $|A| \leq |R(A)|$ for every subset of A of X, where $R(A)$ denotes the set of vertices in Y that are adjacent to vertices in A.*

A matching in a bipartite graph is as selection of edges such that no two edges are incident with the same vertex, and a complete matching from X to Y in a bipartite graph is a matching in which there is an edge incident with every vertex in X. For example, there is a complete matching from X to Y in Figure 2.8 (a), but not in Figure 2.8(b), since $|\{x_2, x_3\}| = 2 > |R(\{x_2, x_3\})| = |\{y_4\}| = 1$ in Figure 2.8 (b).

Figure 2.9 (b) illustrates Lemma 2.3 when $n = 5$ and $i = 1$.

Theorem 2.5 *Let SC be the set of all sets with the C property. Then, $min_{S \in SX}\{|S|\} = C^n_{\lfloor n/2 \rfloor}$, where C stands for combination.*



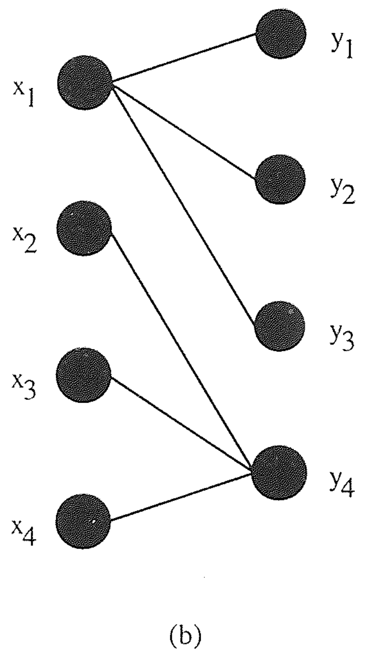
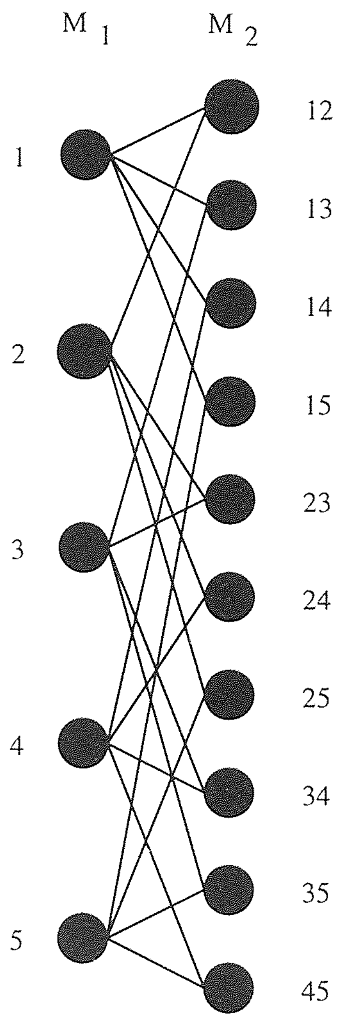
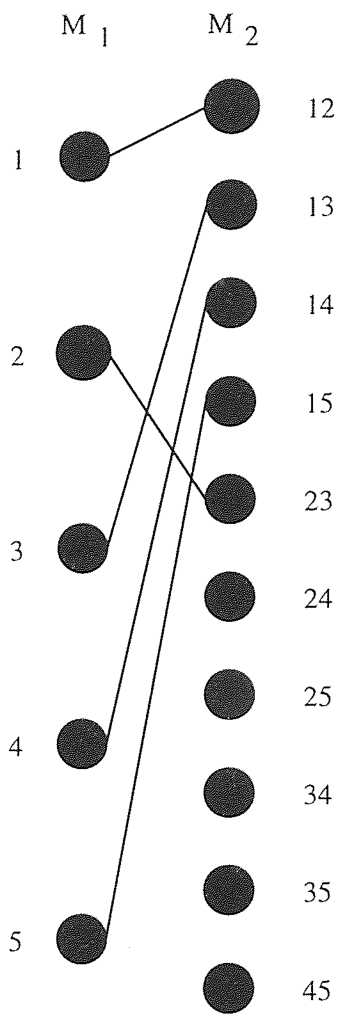


Figure 2.8 Example of illustrating the Theorem of Matching.



(a)



(b)

Figure 2.9 A complete matching from M_1 to M_2.

An example of determining the GC's required for complete subcube recognition in a Q_5 is given in Figure 2.10. The method introduced in the proof of Theorem 2.4 has placed arcs from M_i to M_{i-1}, $2 \leq i \leq n$, in Figure 2.10 (a), and the procedure of determining the required GC's is shown in Figure 2.10 (b).

Let $\alpha(n)$ be the minimal number of GC's required for complete subcube recognition in a Q_n. Then, the following corollary follows from Theorem 2.4.

Corollary 2.1 $\alpha(n) \leq C^n_{\lfloor n/2 \rfloor}$.

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

Proposition 2.1 $\{\Pi^n_{k=1}(2k-1)\}/(2^n n!) < (2n+1)^{-1/2} \forall n \in I^+$.

From $C^n_{\lfloor n/2 \rfloor} = \Pi^{\lceil n/2 \rceil}_{k=1}{}_{k=1}(2k - 1/k)2^{\lfloor n/2 \rfloor}$ and the above proposition, it can be verified that $\lim_{n\to\infty} C^n_{\lfloor n/2 \rfloor}/2^n = 0$ and $\lim_{n\to\infty} C^n_{\lfloor n/2 \rfloor}/a^n = \infty \ \forall a \in [0, 2)$, meaning that the complexity of $C^n_{\lfloor n/2 \rfloor}$ is still exponential. However, the above result bears practical importance, since the number of the GC's required for complete subcube recognition is significantly reduced according to Corollary 2.1; especially, this is true when n is large because $\lim_{n\to\infty} C^n_{\lfloor n/2 \rfloor}/2^n = 0$.

2.2.4 Maximal Set of Subcubes (MSS)

The MSS strategy was proposed by Dutt and Hayes [1] in 1991. This strategy is based on the notion of a Maximal Subset of Subcubes (MSS), which is a set of disjoint subcubes composed of all of the available processors in the hypercube that has the property of being greater than or equal to all other sets of disjoint subcubes composed of all of the available processors. As described by Dutt and Hayes [1], a set of subcubes is considered greater than another ($A > B$) if the following conditions exist.

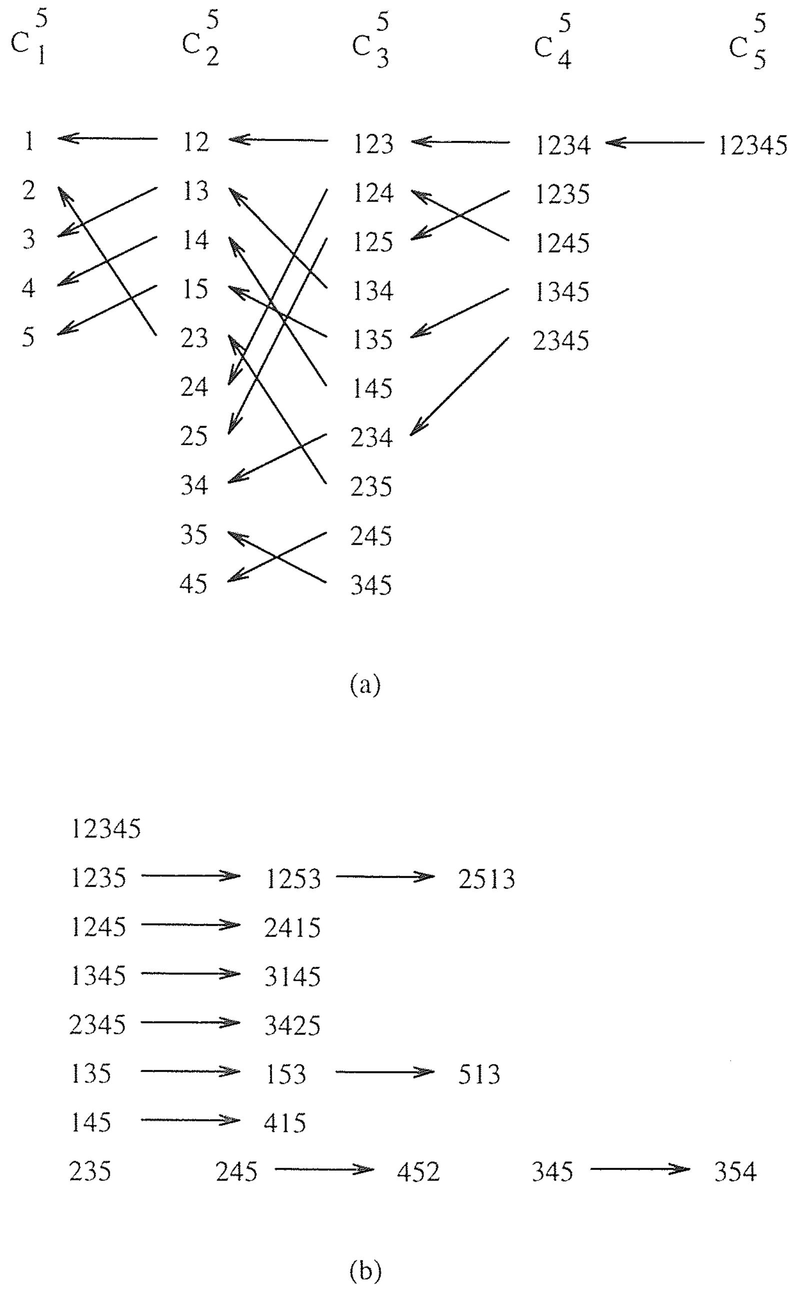


Figure 2.10 The GC's required for complete subcube recognition in a Q_5. (a) Determime the GC's for complete subcube recognition. (b) Modification of GC's in (a).

1) A and B contain the same number of processors.

2) There exists an integer k, $0 =< k =< n$, such that for all m, $k =< m =< n$, A and B have equal numbers of subcubes of dimension m, and A has more subcubes of dimension k than B.

The goal of the MSS strategy is to maintain the greatest MSS as a free-list of available processors after every allocation and deallocation on a subcube. A subcube can then be allocated directly out of this list if the list contains a subcube of the requested dimension. If it does not, but it contains larger subcubes, a larger subcube is chosen and decomposed in such a way that the greatest MSS is left behind. The major portion of the overhead of the MSS strategy arises from searching for this best decomposition, because all possible decompositions must be considered. This decision problem is NP-hard with time complexity $O(2^{3^n})$. The MSS strategy goes beyond the complete subcube recognition provided by MGC, because it not only finds a subcube of a given size if one exists but also chooses the one whose allocation will leave behind the greatest MSS.

Note that we basically have two forms of MSS-based subcube allocation strategies, MSS_STRATEGY and FAST_MSS_STRATEGY that use BEST_FIT and HUERISTIC_COALESCE, Dutt and Hayes [1]. The primary difference between these two strategies is that when a k-cube cannot be allocated by BEST_FIT, and there are at least 2^k free nodes, MSS_STRATEGY forms an approximate MSS S, and again checks to see if there is a k-cube in S, while FAST_MSS_STRATEGY skips this step when a request cannot be allocated by BEST_FIT.

We now argue that the simple allocation scheme BEST_FIT is actually quite effective in returning a good approximation of the greatest MSS obtainable after an allocation.

Theorem 2.6

a) Any allocation of a k-cube from the current MSS to an incoming k-cube request yields the greatest MSS that can be obtained after allocating a k-cube.

b) If there are no k-cubes in the current MSS, and m is the dimension of the smallest cube in MSS ($> k$), then no MSS obtained after allocating k-cube by splitting an m-cube in the current MSS can have more than one cube of each of the dimensions $m-1, m-2, \cdots, k+1$.

In case b) of Theorem 2.6, it is possible that the greatest MSS after allocation of a k-cube can obtain more than one k-cube. When BEST_FIT is used to make allocations from an MSS in such a situation, it returns a Set of Free Subcubes (SFS), Dutt and Hayes [1], that has exactly one cube of each of the dimensions $m-1, m-2, \cdots, k$, which is thus a good approximation of the greatest MSS. For case a), BEST_FIT returns the greatest MSS possible. Extrapolating this argument, we can state that when BEST_FIT is used to make allocations from a good approximation S of the MSS, it always returns a good approximation of the greatest MSS obtainable after allocating k-cube.

It is easy to see that the worse case time complexity of BEST_FIT is $O(n)$. Thus, the complexity of HEURISTIC_COALESCE, Dutt and Hayes [1], is the dominating factor in FAST_MSS_STRATEGY, whose complexity is thus $O(n2^n)$. In practice, however, as shown by [1], the time taken by FAST_MSS_STRATEGY is reasonably small. The worse case complexity of MSS_STRATEGY is greater due to its use of APPROX_MSS, Dutt and Hayes [1]; however, once more, as shown by Dutt and Hayes [1], its actual execution time is small. It should be noted that the worst case complexity of BUDDY_STRATEGY and the SGC method, Chen and Shin [6] is $O(2^n)$, while that of the MGC method, Chen and Shin [6] is $O(2^n\binom{n}{\lfloor n/2 \rfloor})$. Furthermore, neither of these allocation strategies attempts to reduce fragmentation of the hypercube, whereas MSS_STRATEGY and FAST_MSS_STRATEGY do so to a certain extend, due to the fact that they always maintain a good approximation of the MSS.

CHAPTER 3

PARTITIONABLE MULTI PROCESSOR ALLOCATION ALGORITHM

3.1 Formal Description of the Problem

In an n-dimensional hypercube (n-cube), multiprocessor, a task is viewed as a set of interacting modules, which must be assigned to a cube. Thus, processor allocation in an n-cube multiprocessor consists of two sequential steps. The first one is to find the number of resources that should be allocated on a multiprocessor for running an application program. An application program/algorithm is represented by a number of interacting modules where each module can be assigned to a processing node of a hypercube. The number of nodes required for a task (job/algorithm) depends on the task flow graph. It has been reported that some regular interconnection topologies such as the ring, tree, and mesh can be embedded on a hypercube, Saad and Schultz [5]. This implies that if we know the size of a topology, the subcube size for accomodating the task is known. Of coarse, not all topology sizes are valid, since the proposed algorithm *Allocation Strategy I (ASI)* does not allow external fragmentation. So, for certain topologies which are not possible we might have some internal fragmentation, which at maximum, we will have three processors unallocated. Hence, we assume that the size of the subcube for an incoming request is known.

The second step in the processor allocation is to locate and assign the required number of resources, as required by a task, on a multiprocessor. On a hypercube, this problem reduces to finding and allocating an appropriate subcube in the machine. This second step of the processor allocation scheme on a hypercube is addressed in this thesis. Although this is the second part of the two-step process, we would call it *processor allocation* without loss of generality.

An efficient processor allocation scheme maximizes the resource utilization, reduces external as well as internal fragmentation, and finally improves system performance. An allocation policy is called *static* if the incoming requests are considered for allocation only at some specific time intervals. It does not consider deallocation (processor relinguishment) at any arbitrary time. On the other hand, a *dynamic* policy can handle processor allocation and deallocation at any time depending on the arrival and completion of jobs. A dynamic policy gives better utilization of resources than a static allocation. However, finding a perfect dynamic policy at minimal overhead is extremely hard. Furthermore, the allocation problem becomes more difficult when some specific nodes must be allocated and/or excluded for some specific tasks. The inclusion situation occurs when some resources are reachable only through specific nodes. The exclusion problem arises when some nodes are faulty or are designated for other purposes and, therefore, cannot be allocated.

3.2 Allocation Strategy I (ASI)

This section describes the allocation strategy *ASI* that we propose, which has the ability to allocate not only complete cubes, but also incomplete ones.

One possible approach in implementing this allocation is to assign a large cube of dimension m that can accommodating the y nodes, i.e., $m \geq \lceil \log y \rceil$, and deallocate the unnecessary $(2^m - y)$ nodes. If an m-cube is not possible, the number of nodes y is divided into smaller subcubes. For example, if a task needs 11 nodes, then the request is divided into 8 and 4-node requests. The 8 nodes (3-cube) and the 4 node (2-cube) are allocated such that they are adjacent. Instead of allocating a 2-cube, one can allocate a 1-cube and a 0-cube to make the number of nodes exactly 11. This thesis focuses on allocating exact number of nodes, and it is achieved by using the (cyclic) BRGC. Under this algorithm, the worst-case time complexity of both allocation and deallocation is $O(n)$, where n is the dimension of the hypercube containing 2^n processors.

Definition 3.1 *The (cyclic) binary reflected gray code of n bits BRGC(n) is defined recursively by BRGC(n)=$\{$0$\bullet$BRGC($n-1$),1$\bullet$BRGC^{-1}($n-1$)$\}$, where "$\bullet$" denotes concatenation, BRGC^{-1}($n-1$) denotes the sequence derived by reversing the order of elements in the sequence BRGC($n-1$), and BRGC(1)=$\{$0,1$\}$.*

Example 3.1 *BRGC(2)=$\{$00,01,11,10$\}$ and BRGC(3)=$\{$000,001,011,010,110,111, 101,100$\}$.*

Theorem 3.1 *Given the request of size y, assuming $y \leq 2^n$, the ASI algorithm guaranteed that the maximum number of hops is given by $\lceil log(m) \rceil$, where m is the smallest possible cube that the incomplete y-cube fits in.*

Proof. It has been known that the maximum number of hops in an m-cube is equal to $log(m)$. Since $y \leq 2^m$, by taking the ceiling of the $log(m)$ ($\lceil log(m) \rceil$), it is garanteed that it will be equal to the maximum number of hops. ■

Theorem 3.2 *The remaining r nodes that have not yet been allocated, are gurandeed to be multiples of 4, and can form at maximum a complete 5-dimensional hypercube.*

Proof. Due to the restrictions in our algorithm, we do not allocate any processors unless the remaining r nodes are multiples of 4. Furthermore, our algorithm requires that the remaining r processors form complete cubes of maximum size of 5. Thus, Theorem 3.2 holds. ■

Any gray code has the property that any two neighboring codes in the sequence of all possible 2^n n-bit numbers differ in a single bit. The cyclic version of the BRGC(n) is used throughout this thesis. The presentation of the ASI is very detailed for the purpose of clarity, and the steps are as follows.

Processors Allocation

Step 1. Give the size of the n-cube, and the request of size y.

Step 2. Calculate the number p, of incomplete y-cubes that can be allocated from the n-cube, where $p = (\lfloor 2^n/y \rfloor)$, and y-cube=incomplete cube with y nodes.

Step 3. Verify that by allowing these p possible incomplete cubes, the number of remaining r nodes, is a multiple of 4. Algorithmically, this can be verified by forming the r mod 4 function and the outcome should be equal to 0, ($r\%4 \overset{?}{=} 0$).

Step 4. If the remaining nodes r, are not a multiple of 4, then increase the value of the request size y by 1, and go back to step 2; otherwise, skip this step and go to step 5. (*Note*, that steps 2 and 3, might be repeated at maximum 3 times, thus, having at 3 nodes unallocated. If all the above steps are satisfied, then we do the allocation.)

Step 5. Generate the address space of the n-cube using the *BRGC(n)*.

Step 6. The addresses for the p y-cubes are generated using the following algorithm. Assuming that we number the y-cubes from 0 to $p-1$. Let Y_0 be the address range for the 0^{th} y-cube, and in general Y_n is the address of the Y_{n-1} y-cube. Algorithmically,

$$Y_0 = 0 \text{ to } A_0,$$
$$Y_1 = A_0 + 1 \text{ to } A_0 + 1 + y$$
$$\vdots$$
$$Y_{p-1} = A_{p-2} + 1 \text{ to } A_{p-2} + 1 + y$$
$$Y_{p-1} = A_{p-1} + 1 \text{ to } A_{p-1} + 1 + y$$

where $A_0 = bin(y-1)$, and $bin(x)$ is the binary representation of x.

Step 7. Let g be the number r of unallocated nodes in the n-cube.

Step 8. If $(g/temp) \neq 0$, it means that more than $temp$ nodes (at first $temp = 32$), are available, therefore allocate sequentially starting from address $A_p + 1$ a 5-cube (a 5-dimensional hypercube). Skip to step 10.

Step 9. If $(g/temp) = 0$, it means that the less than $temp$ nodes (at first $temp = 32$), are available, so reduce the size of $temp$ nodes ($32 \rightarrow 16 \rightarrow 8 \rightarrow 4$), and go back to step 8. (*Note*, that if $temp = 4$, it means that these 2-cube are the last nodes available from the n-cube and must exit the terminate procedure.)

Step 10. Reduce the size of $temp$ nodes ($16 \rightarrow 8 \rightarrow 4$), and go back to step 8. (Once again if at this stage $temp = 4$, exit the terminate procedure.)

Example 3.2 *Let us assume that* $n = 5$ *and a request of size* $y = 7$ *comes along. We calculate the number* p, *of incomplete* 7*-cubes that can be allocated from the complete* 5*-cube, by* $p = (\lfloor 2^n/y \rfloor) = (\lfloor 2^5/7 \rfloor) = (\lfloor 32/7 \rfloor) = 4$. *We then verify that by allowing* 4 *possible incomplete cubes, the number of the remaining* r *nodes, is a multiple of* 4, $r = (2^5 - 4*7) = (32-28) = 4 \Rightarrow r/4 = 4/4 = 0$. *Then generate the address space of the* 5*-cube using the BRGC(5). We let* g *be the number* $r = 4$ *of the unallocated nodes in the* 5*-cube. Then let* $temp = 32$ *and check if* $g/temp = 4/32$ *is 0 or not. In this case* $g/temp = 0$, *thus the unallocated nodes are indeed less than 32 so we decrease* $temp = 16$. *Once again* $g/temp = 0$, *so we decrease* $temp = 8$, *and check if* $g/temp$ *is* $= 0$, *which it is, thus* $g = 4 < 8$. *Then decrease temp to its smallest possible value* 4. *In this case* $g/temp = 4/4 \neq 0$ *it is 1, thus we can allocate sequentially a* 2*-cube, and thus we have allocated all the nodes of the* 4*-cube and we stop. The final allocation of the nodes is given graphically in Figure 3.1, and bitwise in Table 3.1.*

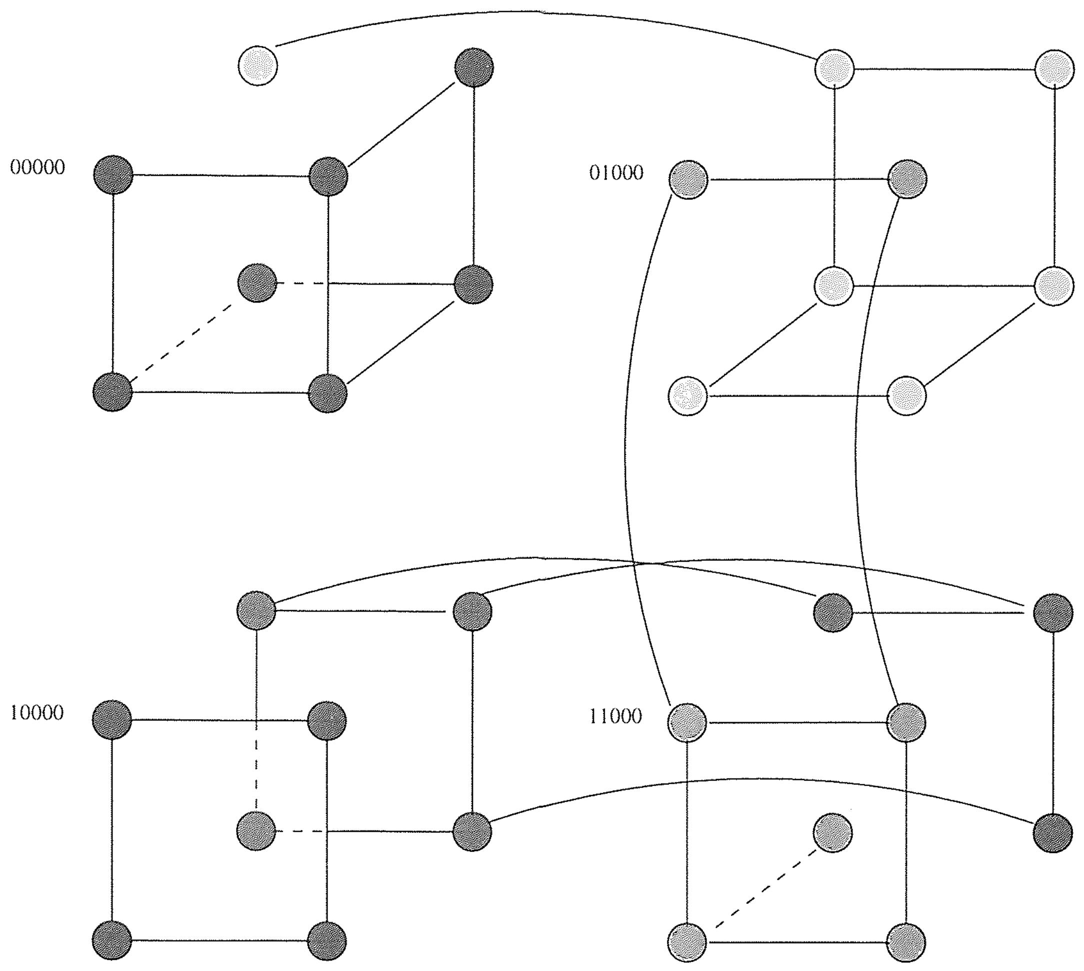


Figure 3.1 Illustration of Example 3.2

Table 3.1 Illustration of bitwise allocation of Example 3.2

Partitions				
1	2	3	4	5
00000	00100	01001	11111	10010
00001	01100	01000	11101	10011
00011	01101	11000	11100	10001
00010	01111	11001	10100	10000
00110	01110	11011	10101	
00111	01010	11010	10111	
00101	01011	11110	10110	

Example 3.3 *Let us assume that* $n = 6$ *and a request of size* $y = 22$ *comes along. We calculate the number* p*, of incomplete* 22*-cubes that can be allocated from the complete* 6*-cube, by* $p = (\lfloor 2^n/y \rfloor) = (\lfloor 2^6/22 \rfloor) = (\lfloor 64/22 \rfloor) = 2$*. We then verify that by allowing* 2 *possible incomplete cubes, the number of the remaining* r *nodes, is a multiple of* 4*,* $r = (2^6 - 2*22) = (64-44) = 20 \Rightarrow r/4 = 20/4 = 0$*. Then generate the address space of the* 6*-cube using the BRGC(*6*). We let* g *be the number* $r = 20$ *of the unallocated nodes in the* 6*-cube. Then let* $temp = 32$ *and check if* $g/temp = 22/32$ *is 0 or not. In this case* $g/temp = 0$*, thus the unallocated nodes are indeed less than 32 so we decrease* $temp = 16$*. Now* $g/temp = 20/16 \neq 0$*, thus, the unallocated nodes are more than 16 so we allocated sequentially a* 4*-cube. Then we update the number* g *of the unallocated nodes by letting* $g = r - 16 = 20 - 16 = 4$*. We also decrease* $temp = 8$*, and check if* $g/temp$ *is* $= 0$*, which it is, thus* $g = 4 < 8$*. Then temp is decreased to its smallest possible value* 4*. In this case* $g/temp = 4/4 \neq 0$*, thus we can allocate sequentially a* 2*-cube, and we have allocated all the nodes of the* 4*-cube and the algorithm stops. The final allocation of the nodes is given graphically in Figure 3.2.*

Table 3.2 Illustration of bitwise allocation of Example 3.3

Partitions			
1	2	3	4
000000	011101	111010	100010
000001	011100	111011	100011
000011	010100	111001	100001
000010	010101	111000	100000
000110	010111	101000	
000111	010110	101001	
000101	010010	101011	
000100	010011	101010	
001100	010001	101110	
001101	010000	101111	
001111	110000	101101	
001110	110001	101100	
001010	110011	100100	
001011	110010	100101	
001001	110110	100111	
001000	110111	100110	
011000	110101		
011001	110100		
011011	111100		
011010	111101		
011110	111111		
011111	111110		

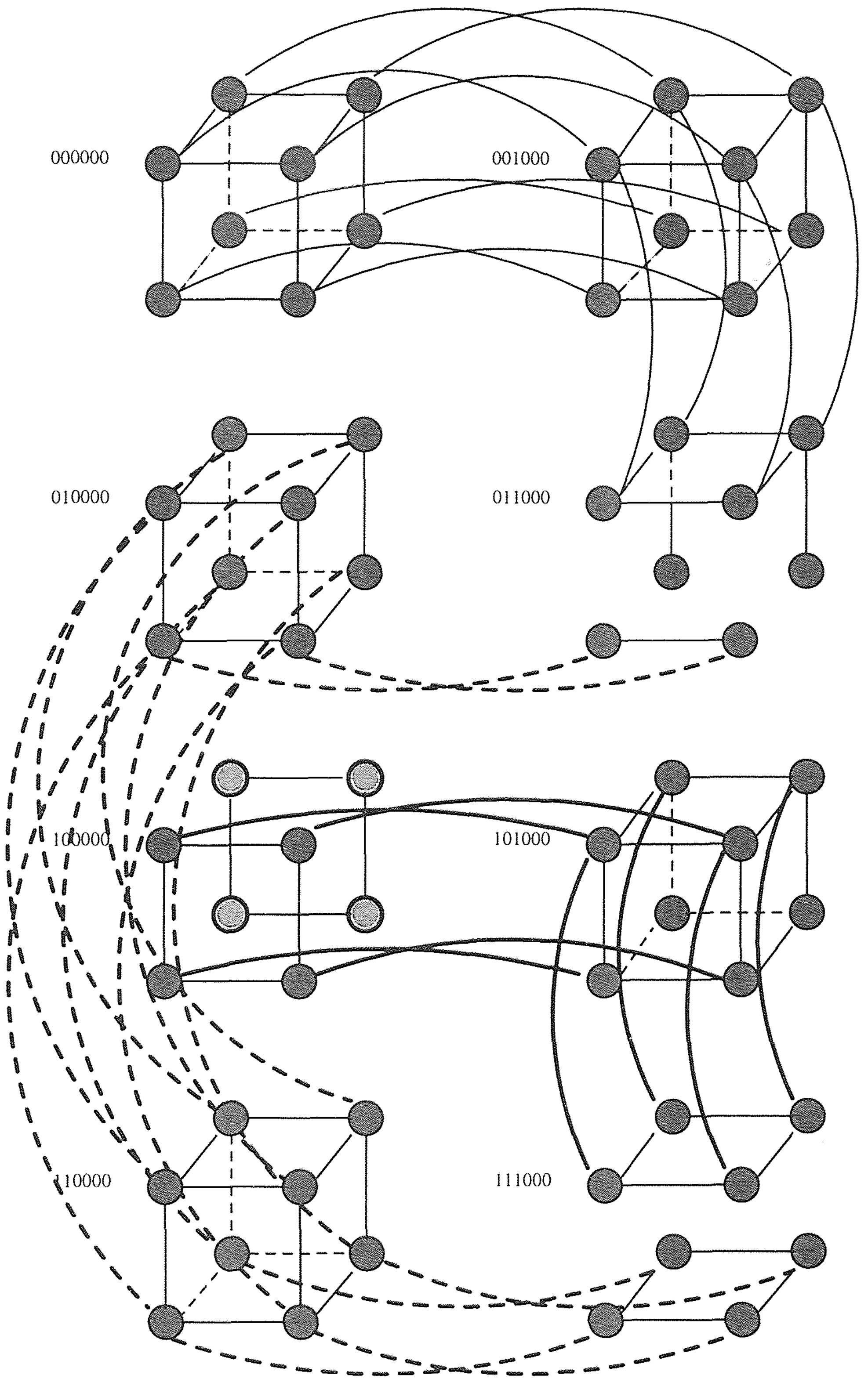


Figure 3.2 Illustration of Example 3.3

CHAPTER 4

SIMULATION RESULTS AND DISCUSSION

In this chapter, we will show how the *ASI* outperforms the existing allocation algorithms by using 3 simple examples illustrated in numerical and graphical forms.

For the first simulation, let us assume that we have available a 5-cube, (thus 32 processors), and the requests are for cubes with 7 processors. All of the existing allocation strategies are able to allocate 4 partitions of 8 processors (3-cube), and thus we will have 1 processor in each partition that will not be allocated. In the case of the *ASI*, we are able to allocate 4 partitions of 7 processors each, thus having incomplete cubes, and the remaining 4 processors are grouped together to form a 2-cube. Therefore, *ASI* allocates all the processors without having any unused ones.

Figure 4.1 shows how the buddy, SGC, MGC and MSS allocation strategies will allocate the 4 partitions, by leaving one processors unallocated in each 3-cube. On the other hand, Figure 4.2 shows how the ASI allocation strategy allocates the 4 partitions of 7 processors, as well as the grouping of the remaining 4 into a 2-cube.

Furthermore, if we go into more complex structures, the ASI algorithm, not only outperforms the existing algorithms by allocating the unused processors to the other tasks, but also can create more partitions. Thus one can decrease the finish time of the parallel tasks.

To show how the ASI can form more partitions than the other algorithms in certain cases, let us assume that we have a 6-cube, and we have requests of 10 processors per partition. The buddy, SGC, MGC and MSS algorithms, are able to allocate 4 partitions of 4-cubes, and thus we will have 6 processors in each partition that will not be used by the other tasks. On the other hand, the ASI algorithm is able to allocate 6 partitions of exactly 10 processors each by forming incomplete cubes.

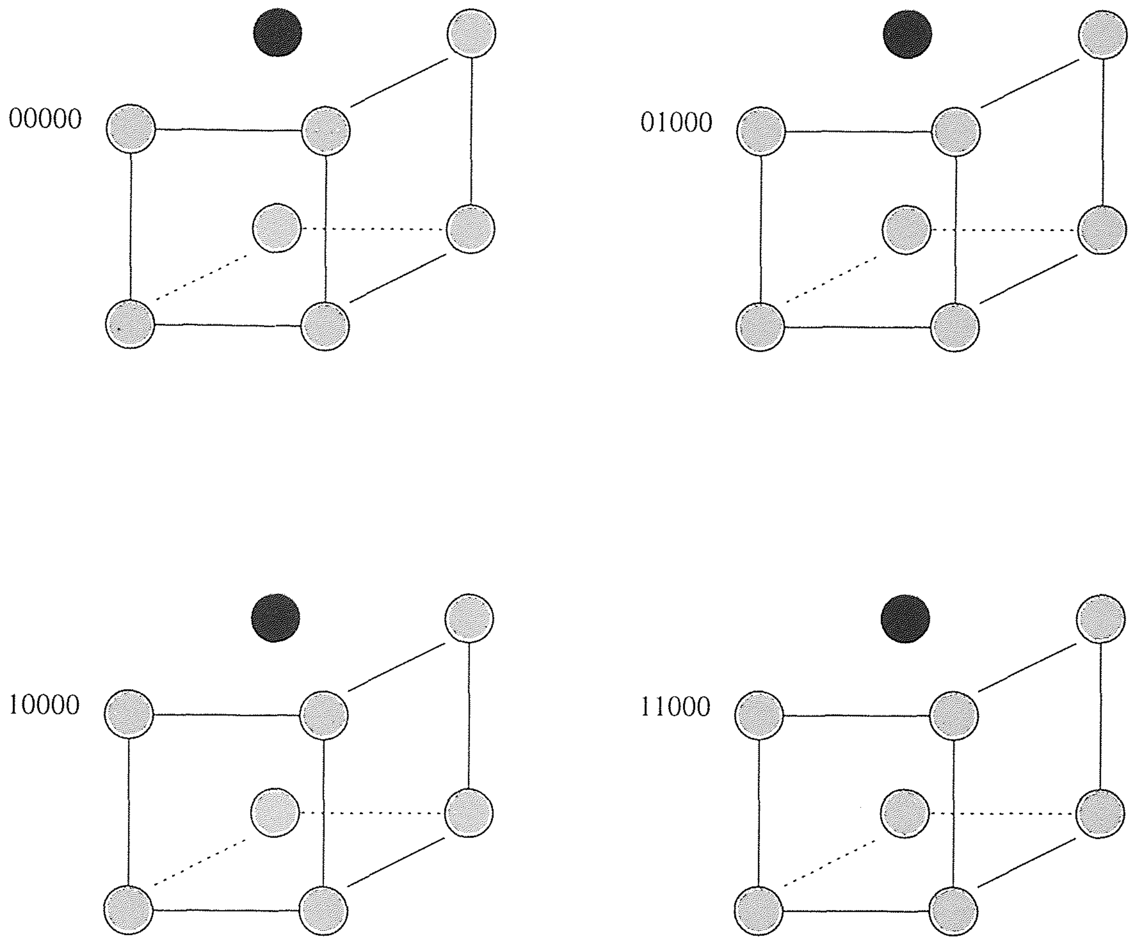


Figure 4.1 Illustration of how Buddy, SGC, MGC and MSS allocate 4 partitions and leaving 1 processor per partition unused.

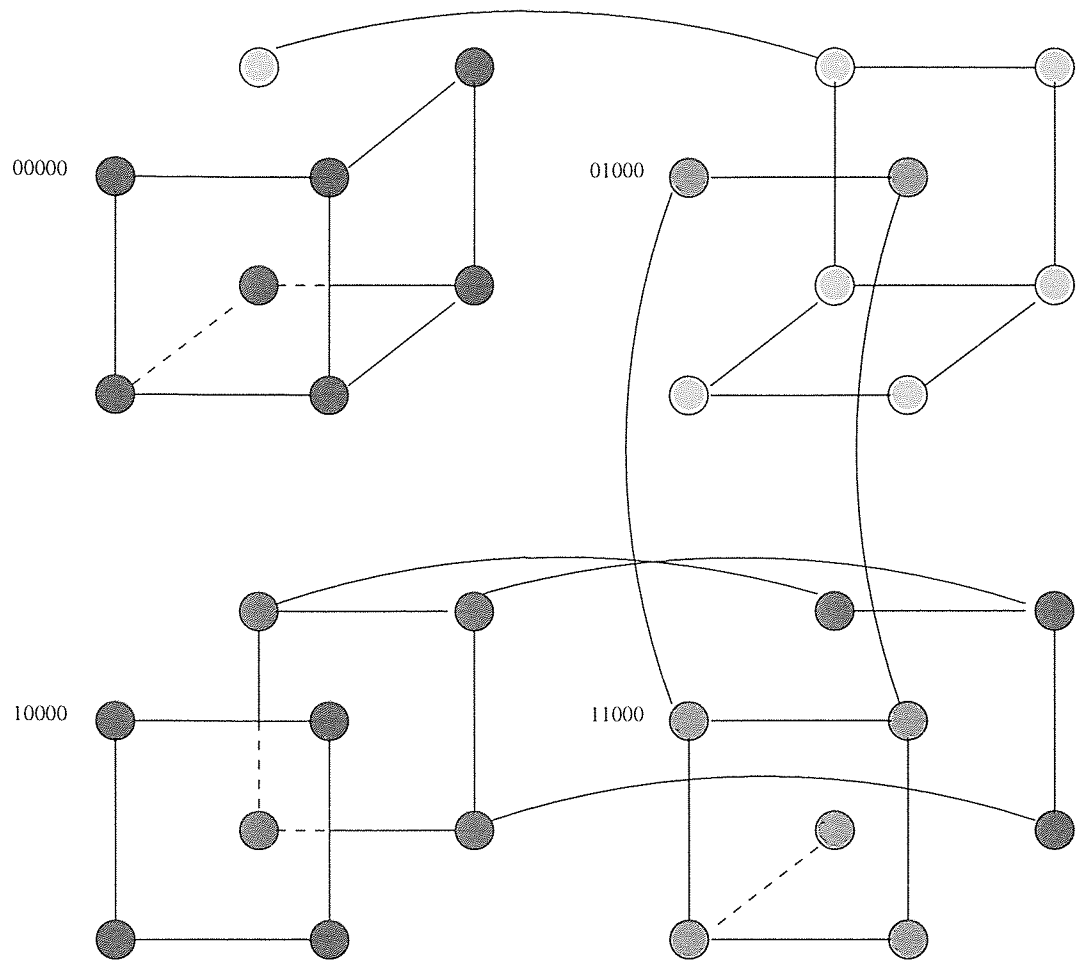


Figure 4.2 Illustration of how ASI allocates 4 partitions and uses the remaining processors to form a 2-cube.

It will also use the remaining processors to form a 2-cube that can be allocated to some other task. If we assume that we had 12 tasks to run, the existing algorithms would require to turn around 3 times, thus allocating 4 4-cubes 3 times. The ASI algorithm, requires only 2 times of turn around, thus having a speedup of 3/2 over the existing algorithms for this siduation.

Figure 4.3 shows how the buddy, SGC, MGC and MSS allocation strategies will allocate the 4 partitions, by leaving 6 processors unused in each 4-cube. On the other hand, Figure 4.4 shows how the ASI algorithm allocates 6 partitions of 10 processors, as well as using the 4 unallocated processors to form a 2-cube.

Furthermore, let us assume that we have a 6-cube, and we have requests of 5 processors per partition. The buddy, SGC, MGC and MSS algorithms, are able to allocate 8 partitions of 3-cubes, and thus we will have 3 processors in each partition that will not be used by the other tasks. On the other hand, the ASI algorithm is able to allocate 12 partitions of exactly 5 processors each by forming incomplete cubes. The latter, also uses the remaining processors to form a 2-cube that can be allocated to some other task. If we assume that we had 24 tasks to run, the existing algorithms would require to turn around 3 times, thus allocating 8 3-cubes 3 times. The ASI algorithm, requires only 2 times of turn around, thus having a speedup of 3/2 over the existing algorithms for this siduation.

Figure 4.5 shows how the buddy, SGC, MGC and MSS allocation strategies will allocate the 8 partitions, by leaving 3 processors unused in each 3-cube. On the other hand, Figure 4.6 shows how the ASI algorithm allocates 12 partitions of 5 processors, as well as using the 4 unallocated processors to form a 2-cube.

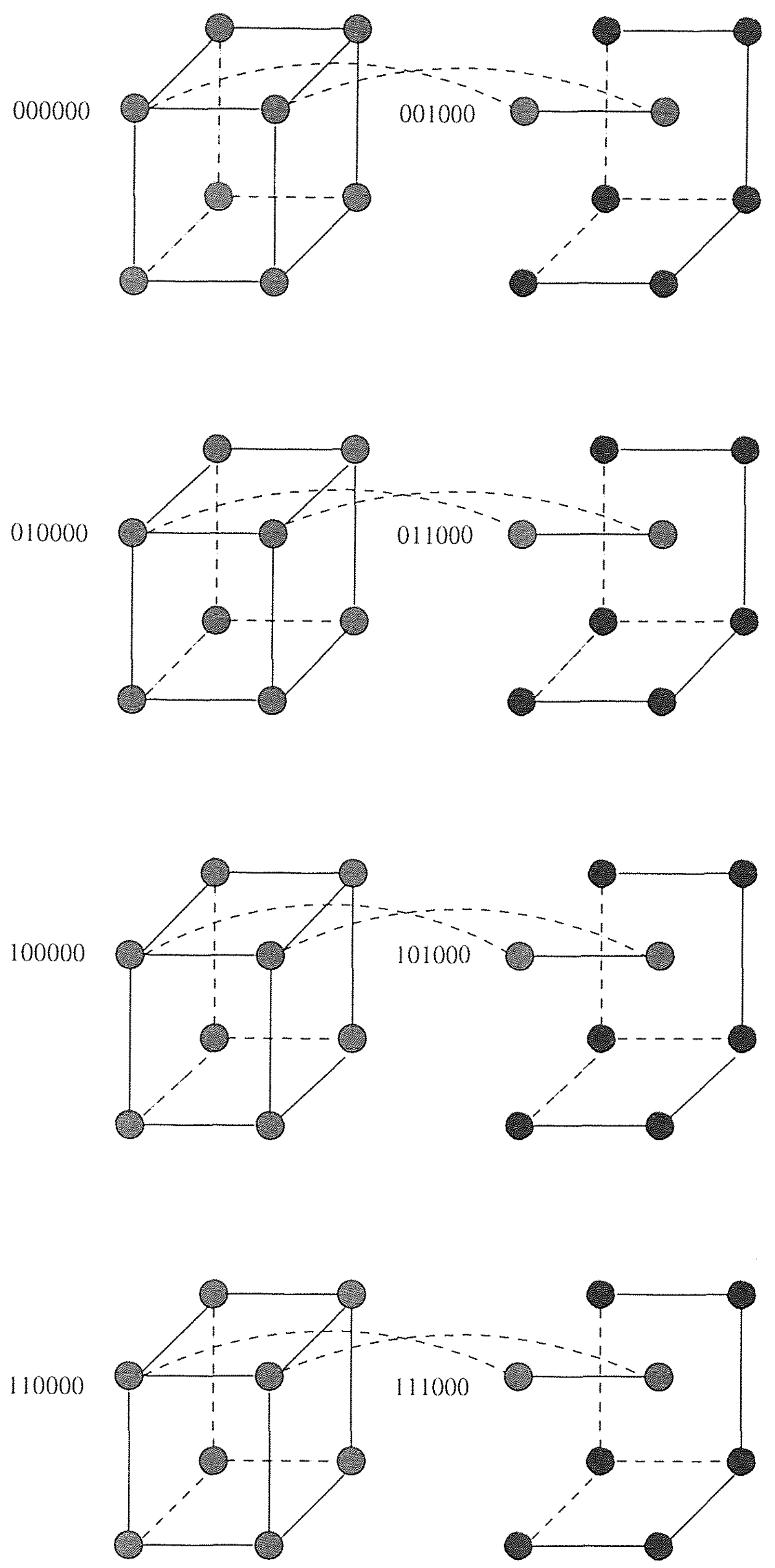


Figure 4.3 Illustration of how Buddy, SGC, MGC and MSS allocate 4 partitions and leaving 6 processors per partition unused.

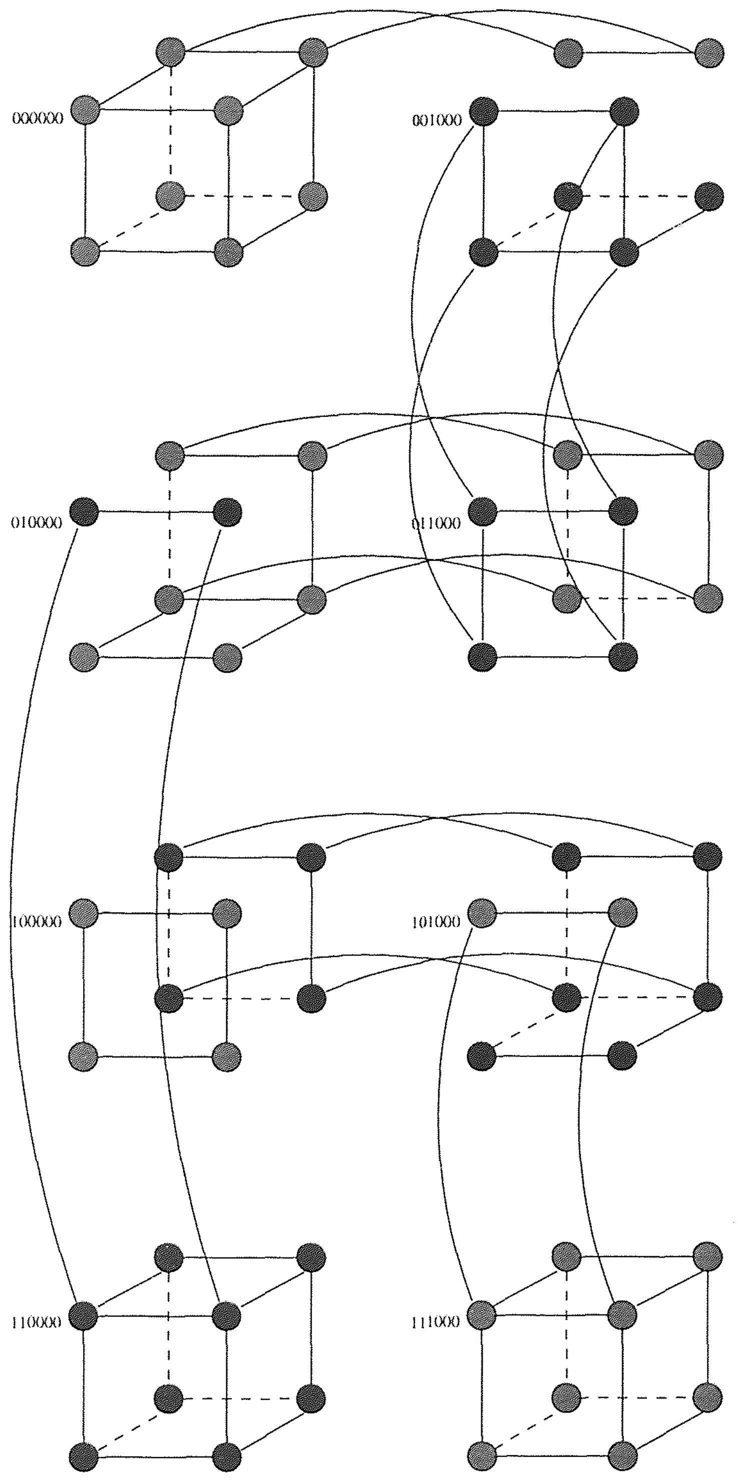


Figure 4.4 Illustration of how ASI allocates 6 partitions and uses the remaining processors to form a 2-cube.

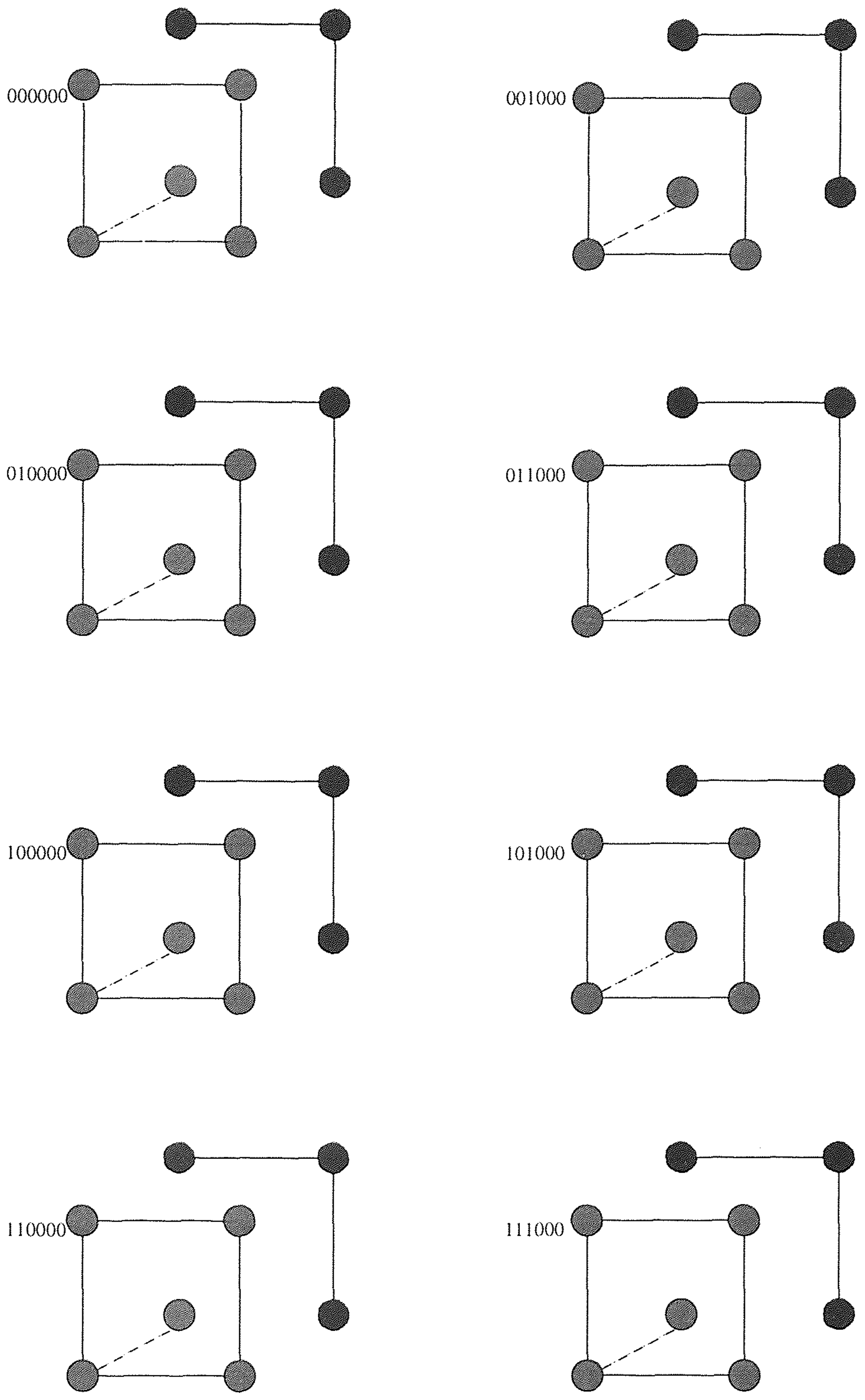


Figure 4.5 Illustration of how Buddy, SGC, MGC and MSS allocate 8 partitions and leaving 3 processors per partition unused.

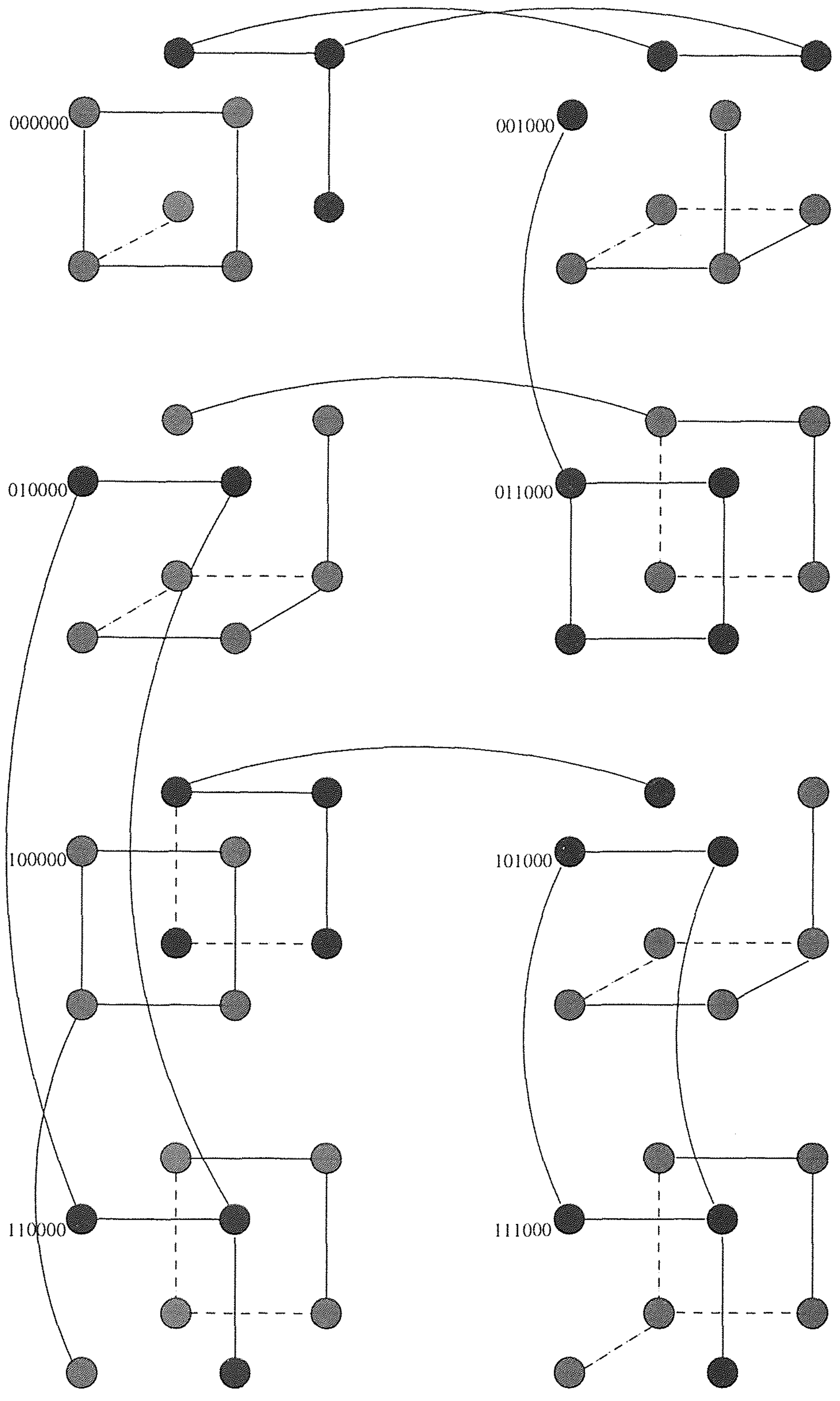


Figure 4.6 Illustration of how ASI allocates 12 partitions and uses the remaining processors to form a 2-cube.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

Efficient processor allocation is essential for achieving high performance on a multiprocessor system. In this thesis, we have first investigated the properties of the buddy, SGC, MGC, and MSS allocation strategies, and then we proposed a new algorithm, *Allocation Strategy I (ASI)*, using the BRGC. The latter outperforms the former by providing better subcube recognition and allocation abilities. We have considered allocation strategies using MGC's, and have seen the relationship between the MGC's used and the corresponding subcube recognition ability. The minimal number of GC's required for complete subcube recognition in a Q_n is found to be less than or equal to $C^n_{\lfloor n/2 \rfloor}$, which is significantly less than $n!$ for a brute-force enumeration.

The existing allocation strategies cannot efficiently support multiple partitions of incomplete cubes, while the proposed ASI not only supports incomplete cubes, but also allocated the unused ones to other tasks. Also, it has been shown to outperform the existing ones in simple parallel problems.

This work can be furthered improved by having the ability to allocate smaller partitions of unused processors, like a 0-cube or a 1-cube.

APPENDIX A

Program

```
/*--------------------------------------------------------*/
/* Program I */
/* Calculates all possible partitions for a specific */
/* Hypercube size. */
/*--------------------------------------------------------*/

/* #define */
/*------- */
#define N 6
#define TPN 64
#define z 5


/* header files */
/*-------------*/
#include <stdio.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>


void main ()
{

int i, j, k, n;
int x, y;
int p, r, t;
int a2, b2, c2;
int a3, b3, c3;
int a4, b4, c4;
int a5, b5, c5;
int rgc[TPN][N] ;
int cntr=0, cntrp=0, tmp1;

FILE *fp1 ;
```

```
if((fp1=fopen("data.txt","w"))==NULL)
{
   printf("could not open file data.txt \n") ;
   exit(1) ;
}

system("clear") ;

/*--------------------------------------*/
/* calculate and verify partitions */
/*--------------------------------------*/
printf("Starting calculations . . .") ;

fprintf(fp1,"\n") ;
fprintf(fp1,"Results from Algorithm I\n") ;
fprintf(fp1,"========================\n") ;
fprintf(fp1,"\n") ;

j=N ;
{

         n=pow(2,j) ;
         k=(n/4) ;

         fprintf(fp1,"available processors: %d\n", n) ;
         fprintf(fp1,"maximum number of partitions: %d\n", k) ;

         i=z ;
         {
                  p=(n/i) ;
                  if(p==0)
                     printf("invalid request\n") ;
                  r=(n%i) ;

                  a5=32 ;
                  b5=(r%a5) ;
                  c5=(r/a5) ;

                  a4=16 ;
                  b4=(b5%a4) ;
                  c4=(b5/a4) ;

                  a3=8 ;
```

```
                b3=(b4%a3) ;
                c3=(b4/a3) ;

                a2=4 ;
                b2=(b3%a2) ;
                c2=(b3/a2) ;

                if(b2==0)
                {
                        fprintf(fp1,"%d x %d, ",  p,  i) ;
                        if(c5!=0)
                                fprintf(fp1,"%d x %d, ", c5, a5) ;
                        if(c4!=0)
                                fprintf(fp1,"%d x %d, ", c4, a4) ;
                        if(c3!=0)
                                fprintf(fp1,"%d x %d, ", c3, a3) ;
                        if(c2!=0)
                                fprintf(fp1,"%d x %d, ", c2, a2) ;
                        fprintf(fp1,"\n") ;
                }
        }
        fprintf(fp1,"--------------------\n") ;

}

printf(" done. \n") ;
/*--------------------------------*/


/*--------------------------------*/
/* allocate complete and incomplete cubes */
/*--------------------------------*/
printf("Starting allocation . . . . . ") ;


/* create rgc */
create_rgc(rgc);

fprintf(fp1,"\n") ;
fprintf(fp1,"  Processor Allocation  \n") ;
fprintf(fp1,"======================\n") ;
for(i=0;i<TPN;i++)
{
        fprintf(fp1,"           ") ;
```

```
            for(j=0;j<N;j++)
            {
                    fprintf(fp1,"%d",rgc[i][j]) ;
            }
            fprintf(fp1," \\\\ \n") ;
            cntr=cntr++ ;
            if(cntr==z)
            {
                    fprintf(fp1,"0 ----------\n") ;
                    cntr=0 ;
                    cntrp=cntrp++ ;
            }
            if(cntrp==p)
            {
                    if((cntr==a5)&&(c4!=0))
                            fprintf(fp1,"1 ----------\n") ;

                    if((cntr==a5*c5+a4*c4)&&(c3!=0))
                            fprintf(fp1,"2 ----------\n") ;

/*                  if((cntr==a4)&&(c5==0))
                            fprintf(fp1,"3 ----------\n") ;
*/
                    printf("\n");
                    printf("cntr=%d\n",cntr);
                    printf("c5=%d, c4=%d, c3=%d\n");

                    if((cntr==a5*c5+a4*c4+a3*c3))
                            fprintf(fp1,"4 ----------\n") ;

/*                  if((cntr==a3)&&(c4==0)&&(c5==0))
                            fprintf(fp1,"5 ----------\n") ;
*/
                    if(cntr==a5*c5+a4*c4+a3*c3+a2*c2)
                            fprintf(fp1,"6 ----------\n") ;

/*                  if((cntr==a2)&&(c3==0)&&(c4==0)&&(c5==0))
                            fprintf(fp1,"7 ----------\n") ;
*/
            }
    }
    fprintf(fp1,"=======================\n") ;
    fprintf(fp1,"\n") ;
```

```
printf("done.\n") ;
/*--------------------------------*/


fclose(fp1) ;

}      /* end of algorithm */
/*--------------------------------*/

/*
=================================
functions
=================================
*/

create_rgc(rgc)
int rgc[TPN][N] ;

{
          int i=0,j=0,k=0,l=0 ;
          int r_seq ;
          int mid ;
          int reflect_offset ;



          r_seq=TPN ;

          for(i=0;i<N;i++)
          {
                    r_seq=r_seq/2 ;

                    for(j=0;j<r_seq;j++)
                              rgc[j][i]=0 ;

                    for(j=r_seq;j<2*r_seq;j++)
                              rgc[j][i]=1 ;

                    /* reflect */

                    for(k=i;k>0;k--)
                    {
                              mid=( TPN/power(2,k) )-1 ;
                              reflect_offset=1 ;
                    for(l=mid;l>-1;l--)
                    {
                            rgc[mid+reflect_offset][i]=rgc[l][i] ;
                            reflect_offset++ ;

                    }
            }

        } /* for i */


} /* function */

/* ------------------------------------------------------------ */
int power(b,e)
int b;
int e;
{
        int t ;
        t=b;

        for(e--;e;e--)
                b*=t;
        return b ;

}   /* function */

/* ------------------------------------------------------------ */
```

APPENDIX B

Sample Run

```
Results from Algorithm I
==========================

available processors: 64
maximum number of partitions: 16
12 x 5, 1 x 4,
-----------------------

  Processor Allocation
==========================
       000000 \\
       000001 \\
       000011 \\
       000010 \\
       000110 \\
 -----------
       000111 \\
       000101 \\
       000100 \\
       001100 \\
       001101 \\
 -----------
       001111 \\
       001110 \\
       001010 \\
       001011 \\
       001001 \\
 -----------
       001000 \\
       011000 \\
       011001 \\
       011011 \\
       011010 \\
 -----------
       011110 \\
       011111 \\
       011101 \\
      011100 \\
      010100 \\
----------
      010101 \\
      010111 \\
      010110 \\
      010010 \\
      010011 \\
----------
      010001 \\
      010000 \\
      110000 \\
      110001 \\
      110011 \\
----------
      110010 \\
      110110 \\
      110111 \\
      110101 \\
      110100 \\
----------
      111100 \\
      111101 \\
      111111 \\
      111110 \\
      111010 \\
----------
      111011 \\
      111001 \\
      111000 \\
      101000 \\
      101001 \\
----------
      101011 \\
      101010 \\
      101110 \\
      101111 \\
      101101 \\
----------
      101100 \\
      100100 \\
      100101 \\
      100111 \\
      100110 \\
 ----------
       100010 \\
       100011 \\
       100001 \\
       100000 \\
 ----------
=======================
```

REFERENCES

1. S. Dutt and J. P. Hayes, "Subcube allocation in hypercube computers," *IEEE Trans. Comput.,* vo. 40, pp. 341-352, Mar. 1991.

2. J. Kim, C. R. Das, and W. Liu, "A top-down processor allocation scheme for hypercube computers," *IEEE Trans. Parallel Distrib. Syst.,* vol. 2, pp. 20-30, Jan. 1991.

3. S. Dutt and J. P. Hayes, "On allocating subcubes in a hypercube multiprocessor," in *Proc. 3rd Conf. Hypercube,* pp. 801-810, Jan. 1988

4. F. Harary, J. P. Hayes and H. J. Wu, "A survey of the theory of hypercube graphs," *Comput. Math. Appl.,* vol. 15, no. 4, pp. 277-289, 1988.

5. Y. Saad and M. H. Schultz, "Topological properties of hypercube," *IEEE Trans. Comput.,* vol. 37, pp. 867-872, July 1988.

6. M. S. Chen and K. G. Shin, "Processor allocation in an n-cube multiprocessor using gray codes," *IEEE Trans. Comput.,* vol. C-36, no. 12, pp. 1396-1407, Dec. 1987.

7. P. J. Denning, "Parallel computing and its evolution," *Commun. Ass. Comput. Mach.,* vol. 29, pp. 1163-1167, Dec. 1986.

8. B. Becker and H. U. Simon, "How robust is the n-cube?," in *Proc. 27th Annu. Symp. Foundations Comput. Sci.,* pp. 283-291, Oct. 1986.

9. M. S. Chen and K. G. Shin, "Embedment of interacting task modules into a hypercube multiprocessor," *Proc. Second Hypercube Conf.,* pp. 121-129, Oct.1986.

10. J. P. Hayes *et al.*, " A microprocessor-based hypercube suppercomputer," *IEEE Micro,* vol.6, no. 5, pp. 6-17, Oct. 1986.

11. T. F. Chan and Y. Saad, "Multigrad algorithms on the hypercube multiprocessor," *IEEE Trans. Comput.,* vol. C-35, pp. 969-977, Nov. 1986.

12. NCUBE Corp., "NCUBE/ten: An overview," *Beaverton,* OR, Nov. 1985.

13. E. M. Reingolg, J. Nievergelt, and N. Deo, *Combinatorial algorithm*, Englewood Cliffs, NJ, Prentice-Hall, 1977.

14. P. W. Purdom, Jr. and S. M. Stigler, "Statistical properties of the buddy system," *J. Ass. Comput. Mach.,* vol. 17, pp. 683-697, Oct. 1970.

15. C. L. Liu, *Introduction to Combinatorial Mathematics*, New York, McGraw-Hill, 1968.

16. K. C. Knowlton, "A fast storage allocator," *Commun. Ass. Comput. Mach.*, vol. 8, pp. 623-625, Oct. 1965