論文94-31B-12-1

MIMD 하이퍼큐브의 프로세서 할당에 관한 연구

(Processor Allocation Strategy for MIMD Hypercube)

李 昇 勳*, 崔 相 昉**

(Seung Hoon Lee and Sang Bang Choi)

要約

본 논문에서는 압축 그레이코드 그룹(PGG : Packed Gray code Group)을 사용하여 MIMD 하이퍼큐브상에서 프로세서를 할당하는 알고리듬을 제안하였다. n차원 하이퍼큐브에는 C(n, k) 2n-k개의 k차원 서브큐브가 존재하며, PGG방법을 사용하여 프로세서를 할당하는 경우 C(n, k)개의 PGG를 사용하면 모든 k차원 서브큐브를 찾을 수 있다. 또한 k차원 서브큐브에서 가능한 C(n, k)개의 PGG중에서 40%정도만 사용하여도 C(n, k)개 모두를 사용한 것과 같은 할당능 력을 얻음을 시뮬레이션을 통하여 발견하였다.

Abstract

In this paper, we propose a processor allocation algorithm using the PGG(Packed Gray code Group) for the MIMD hypercube. The number of k-D subcubes in an n-cube is C(n, k) 2n-k. When the PGG is employed in the processor allocation, C(n, k) PGG's are required to recognize all the k-D subcubes in an n-cube. From the simulation we find that the capability of processor allocation using only 40% of C(n, k) PGG's is about the same as that of the allocation using all the PGG's.

I. 서 론

빠른 계산과 고장허용(fault tolerance) 능력을 가진 시스템에 대한 요구의 증가로 다중 프로세서 구조(multiprocessor architecture)에 대한 연구가 활발해졌다. 다중 프로세서에서 여러개의 PE(processing element)들은 상호 연결망에 의해 연결된다. 이러한 상호 연결망에는 여러 가지가 있는데 그중 하이퍼큐브(hypercube)[1]는 다양한 병렬처리 알고리듬에 적합한 구조로 인하여 근래에 와서 많이 연구되고 있다. 이미 Cosmic cube[2] Intel iPSC[3] Connection Machine[4] N-cube/10[5] FPS T-series[6]등과 같이 하이퍼큐브 구조를 적용한 시스템들이 개발되어 연구용 또는 상업용으로 사용되고 있다.

* 準會員 ** 正會員 仁荷大學校 電子工學科
(Dept. of Elec. Eng., Inha Univ.)

* 이 연구는 93년도 한국과학재단 연구비지원에 의한 결과임(과제번호: 931-0900-014-1)
接受日字 : 1994年 5月 28日

여러개의 프로그램을 동시에 수행할 수 있는 MIMD(Multiple Instruction stream-Multiple Data stream)[7] 시스템상에서 어떤 작업이 수행되기 위해서는 우선 작업이 수행될 프로세서를 결정하고, 결정된 프로세서상에서 주어진 프로그램을 수행하여야 한다. 또한 프로그램의 수행이 끝나면 그 작업의 수행에 사용되었던 프로세서를 반환하여 다른 작업의 수행에 이용될 수 있도록해야 한다. 이와 같은 과정을 프로세서 할당과 할당해제라고 한다. MIMD 하이퍼큐브 시스템에서는 어떤 작업이 p개의 프로세서를 요구하는 경우 k차원 ($p \le 2^k$를 만족하는 가장 작은 정수 k) 서브큐브(subcube)를 해당 작업에 할당한다. 프로세서 할당 방법에는 버디(buddy)방법[8] 그레이 코드(Gray code)방법[9] 트리 병합(tree collapsing)방법[10] 프리 리스트(free list)방법[11], 그리고 큐브 합병(cube coalesing)방법[12]등이 있다.

버디 방법은 모든 서브큐브의 검색이 불가능하고 프로세서의 할당해제를 고려하지 않은 방법이다. n차원 하이퍼큐브에서 버디 방법을 사용하여 검색가능한 k차원 서브큐브의 수는 2^{n-k}이다. 그레이 코드 방법의 경우 하나의 그레이 코드를 사용하면 $2\ 2^{n-k}$개의 k차원 서브큐브를 검색할 수 있으며, 모든 서브큐브의 검색을 위해서는 C(n, [n/2])개의 그레이 코드를 사용하여야 한다. 이를 다중 그레이코드 방법이라 한다. 여기서 C(n, m)은 n개 중에서 m개를 택하는 조합의 수이고, [n]은 n을 넘지않는 최대 정수를 의미한다. 프리 리스트 방법은 매우 효율적으로 프로세서 할당을 할 수 있으나, 각 차원마다 사용 가능한 서브큐브의 목록을 저장하기 위한 별도의 메모리 공간이 필요하며 할당해제시 최악의 경우 $O(n^3)$라는 시간 복잡도(time complexity)를 갖는다. 트리 병합 방법은 모든 서브큐브의 검색에 필요한 최소의 검색범위를 제공한다. 그러나 검색범위 결정시 트리구조를 사용하여야 하며, 트리를 재배열하는 변환이 필요하여 시간 복잡도가 커진다. 큐브 합병 방법은 필요한 서브큐브의 크기를 큐브 합병이라는 기법을 사용하여, 작게하는 효과를 얻음으로써 할당비트(allocation bit)의 참조횟수를 줄였다.

본 논문에서는 PGG(Packed Gray code Group)를 이용한 새로운 프로세서 할당방법을 제안하였으며, 다중 그레이코드 방법에서 찾지 못한 k차원 서브큐브 검색시 필요한 최소갯수의 그레이코드의 종류를 PGG를 이용하여 찾았다. n차원 하이퍼큐브에는 C(n, k) 2^{n-k}개의 k차원 서브큐브가 존재하며, PGG를 사용하여 모든 서브큐브를 검색하는 경우 C(n, k)개의 PGG가 필요함을 보이고, 시뮬레이션을 통하여 위에서 설명한 기존의 방법과 성능을 비교하였다. 시뮬레이션 결과 다른 방법보다 매우 우수하였으며, C(n, k)개의 PGG중 40%만 사용하더라도 모든 PGG를 사용한 것과 동일한 할당능력을 얻음을 알았다.

본 논문의 내용은 다음과 같이 구성되어 있다. II장에서는 그레이코드와 압축 그레이코드 그룹을 설명하였다. III장은 PGG를 이용한 새로운 프로세서 할당방법을 설명하였으며, IV장에서는 여러 프로세서 할당 알고리듬의 성능을 비교하기 위하여 수행한 시뮬레이션 결과를 보여준다. V장에서 본 논문의 결론을 맺는다.

II. 압축 그레이코드 그룹

1. 그레이코드

그레이코드는 이웃한 코드와의 해밍거리(Hamming distance)가 1인 코드이다. n개의 비트로 이루어진 그레이코드의 종류는 모두 n!개가 존재하는데 대표적인 것이 BRGC(binary reflected Gray code)이다. n 비트 BRGCn은 다음과 같이 재귀적으로 정의된다.[9]

$BRGC_1 = \{0, 1\}$

$BRGC_k = \{(0)BRGC_{k-1}, (1)BRGC'_{k-1}\}, 2 \le k \le n$

(b)$BRGC_k$, b {0, 1} 는 BRGCk를 k 비트의 이진수로 구성된 수열이라고 했을때 $BRGC_k$의 모든 원소의 최상위 비트 좌측에 비트값 b를 추가한 것이다. 예를 들어 (0)$BRGC_1$은 $BRGC_1$, {0, 1}의 각 원소의 좌측에 0을 추가한 {00, 01}이 된다. 또한 $BRGC'_k$는 $BRGC_k$의 원소를 역순으로 나열한 것이다. $BRGC'_1$은 $BRGC_1$, {0, 1}의 순서를 반대로한 {1, 0}이 된다.

이와 같이 $BRGC_n$을 정의하면 n 비트로 나타낼 수 있는 일반적인 그레이코드는 다음과 같이 구할 수 있다. 우선 BRGCn의 각 비트에 좌측으로부터 1, 2, 3, ⋯, n 과 같이 비트 번호를 붙인다. 그러면 n 비트 그레이코드는 $[g_1, g_2, \cdots, g_n]$ $1 \le g_k \le n$ 으로 나타낼 수 있다. 여기서 gk는 $BRGC_n$의 비트 번호 g_k를 좌측으로부터 k번째 위치로 옮기는 것을 의미한다. $BRGC_n$는 비트 위치와 비트 번호가 일치하므로 $[g_1, g_2, \cdots, g_n] = [1, 2, \cdots, n]$과 같이 나타낸다. $[g_1, g_2, \cdots, g_n]$으로 나타낼 수 있는 그레이코드의 종류는 1부터 n까지의 정수를 나열하는 순열의 경우의 수, n!개가 존재한다. $[g_1, g_2, g_3, g_4] = [3, 1, 4, 2]$인 그레이코드를 그림 1에 나타내었다. 그림에서 $g_1 = 3$ 이므로 $BRGC_4$의 비트 번호 3을 첫번째 위치로 옮긴다. 마찬가지로 $BRGC_4$의 비트 번호 1, 4, 2를 각각 2, 3, 4 번째 위치로 옮긴다.

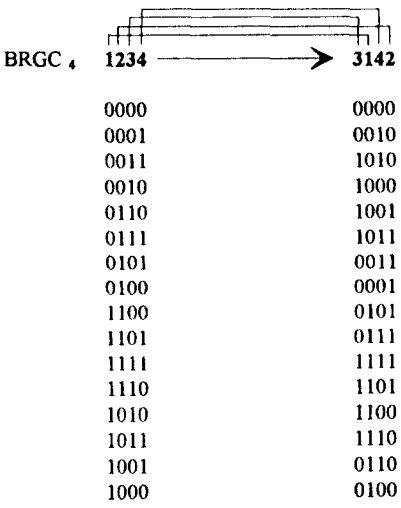


그림 1. [3, 1, 4, 2]그레이코드의 예
Fig. 1. Example of Gray code [3, 1, 4, 2].

2. 압축 그레이코드 그룹 (PGG : Packed Gray code Group)

n차원 하이퍼큐브에서 k차원 서브큐브는 2^k개의 프로세서로 구성된다. 각 프로세서의 번지는 n비트 이진수로 표현되는데, 한 k차원 서브큐브에 속하는 2^k개의 프로세서의 번지는 n-k개의 동일한 비트를 갖는다. 서브큐브의 번지는 n-k개의 동일한 비트를 제외한 나머지 비트를 *(don't care)로 대치함으로써 나타낼 수 있다. 예를 들면 4개의 노드 10001, 10011, 11001, 11011로 구성된 5차원 하이퍼큐브내의 2차원 서브큐브 번지는 1*0*1과 같이 나타난다.

[정의 1]n비트 그레이코드에서 각 원소들은 하이퍼큐브의 프로세서 번지에 해당한다. 그레이코드의 원소들을 처음부터 순서대로 2^k개씩 나누면, 각각은 하이퍼큐브상에서 k차원 서브큐브를 이루는 프로세서들의 집합이 되며, 각 프로세서들의 집합을 서브큐브 번지로 나타낼 수 있는데, 이렇게 그레이코드를 이용하여 서브큐브 번지를 나타낸 것을 압축 그레이코드(packed Gray code)라 하고 $PG^k{}_n$으로 나타낸다. 여기서 아래첨자 n은 압축 그레이코드의 비트수가 n개 임을 나타내며, 윗첨자 k는 2_k개씩 묶어 k차원 서브큐브 번지로 나타냄을 의미한다. 그레이코드 $[g_1, g_2, \cdots, g_n]$을 사용하여 PGk_n를 나타내면 $[g_1', g_2', \cdots, g_n']$가 되며 각 비트 번호 g_i'는 다음과 같이 결정된다.

$$g_i = \begin{cases} g_i, & 1 \le g_i \le n-k \\ *, & n-k+1 \le g_i \le n \end{cases} \qquad \text{(정의 끝)}$$

이것은 주어진 그레이코드 $[g_1, g_2, \cdots, g_n]$에서 비트번호 1부터 n-k까지는 그대로 두고, 비트번호 n-k+1부터 n까지는 *로 바꾸는 것이다. 예를 들어 그레이코드 [3, 1, 4, 2]를 $PG^2{}_4$로 나타내면 [*, 1, *, 2]와 같이 된다.

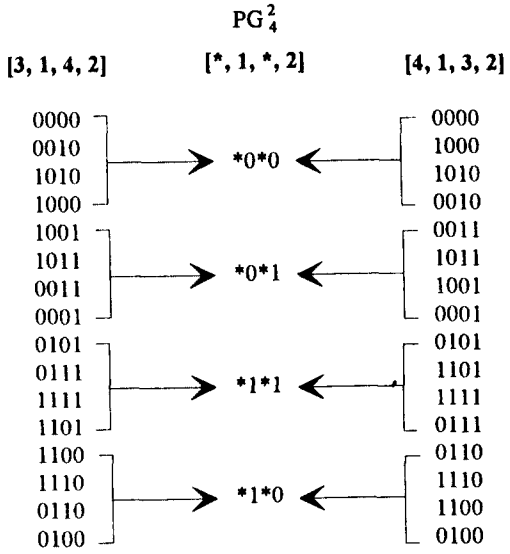


그림 2. 그레이코드 [3, 1, 4, 2]와 [4, 1, 3, 2]를 사용한 $PG^2{}_4$
Fig. 2. $PG^2{}_4$ using Gray codes [3, 1, 4, 2] and [4, 1, 3, 2].

그림 2는 그레이코드 $[g_1, g_2, g_3, g_4]$= [3, 1, 4, 2 [와 [4, 1, 3, 2[를 $PG^2{}_4$로 나타내는 경우 모두 [*, 1, *, 2]가 되는 것을 보여주고 있다. 이것은 두 그레이코드에서 비트 번호 1과 2의 위치는 같고, 비트 번호 3과 4는 $PG^2{}_4$에서는 똑같이 *로 표시되기 때문에 비트 번호 3과 4의 위치를 서로 바꾸더라도 같은 $PG^2{}_4$가 된다. 일반적으로 n비트 그레이코드를 PGkn으로 나타내는 경우 *로 표시된 g_i'를 제외한 나머지 비트번호가 같으면 동일한 $PG^k{}_n$이 된다.

*로 표시된 g_i'에 나열할 수 있는 비트번호의 경우의 수는 k!이 되며, 이렇게 얻어진 k! 개의 그레이코드는 동일한 $PG^k{}_n$로 표현된다. 그리고 n비트로 표현되는 $PG^k{}_n$의 종류는 n개의 비트 위치 중 n-k개를 선택하여 비트 번호 1부터 n-k까지 나열하는 경우의 수 C(n, n-k)·(n-k)!이 된다.

[정의 2] $PG^k{}_n$, $[g_1', g_2', \cdots, g_n']$를 다음과 같이 $g_1'' g_2'' \cdots g_n''$로 나타낸 것을 압축된 그레이코드 그룹(packed Gray code group)이라 하고,

PGGkn로 표기한다.

$$g_i'' = \begin{cases} *, & g_i' = * \\ b, & otherwise \end{cases}$$

$PGG^k{}_n$에서 첨자, n, k는 $PG^k{}_n$를 그룹으로 표시한 것을 의미한다. (정의 끝)

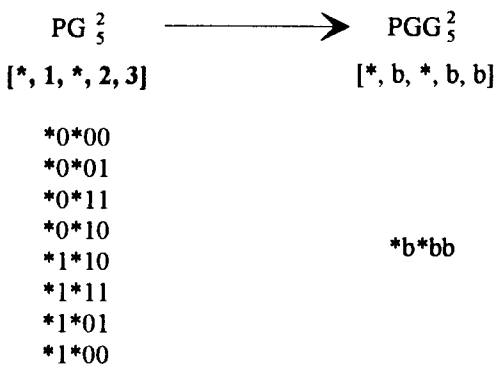


그림 3. $PG^2{}_5$로부터 얻은 $PGG^2{}_5$의 예
Fig. 3. Example of $PGG^2{}_5$ obtained from $PG^2{}_5$.

그림 3은 압축 그레이코드 $PG^2{}_5$로부터 얻은 압축 그레이코드 그룹 $PGG^2{}_5$의 한 예이다. 하나의 압축 그레이코드 그룹, $PGG^k{}_n$은 (n-k)! 개의 $PG^k{}_n$를 포함한다. 예를 들어 6개의 $PG^2{}_5$, [*, 1, *, 2, 3] [*, 2, *, 3, 2] [*, 2, *, 1, 3] [*, 2, *, 3, 1] [*, 3, *, 1, 2] [*, 3, *, 2, 1]을 각각 $PGG^2{}_5$로 나타내면 모두 *b*bb로 동일한 결과를 얻는다. 따라서 *의 위치가 같은 $PG^k{}_n$을 PGGkn로 나타내면 동일한 $PGG^k{}_n$을 얻음을 알 수 있다. 그리고 n비트로 나타낼 수 있는 $PGG^k{}_n$의 종류는 n개의 비트 위치 gi' 중에서 *가 들어갈 k개의 위치를 선택하는 경우의 수이다. 따라서 $PGG^k{}_n$의 종류는 C(n, k)개가 된다.

Ⅲ. $PGG^k{}_n$을 이용한 프로세서 할당 알고리듬

표 1은 4차원 하이퍼큐브에서 가능한 모든 PGG24와 각 $PGG^2{}_4$에 속하는 $PG^2{}_4$, 그레이코드, 그리고 그 $PGG^2{}_4$로 검색할 수 있는 2차원 서브큐브의 번지를 나타낸 것이다. 4차원 하이퍼큐브에서 어떤 작업의 수행에 2차원 서브큐브가 필요한 경우 $PGG^2{}_4$를 사용하여 프로세서를 할당할 수 있다. 표에서 $PGG^2{}_4$, bb**로 검색 가능한 서브큐브는 00**, 01**, 10**, 11**이다. bb**를 사용하여 2차원 서브큐브를 검색한다는 것은 우선 00**내에 포함된 4개의 프로세서, 0000, 0001, 0011, 0010들이 유휴 프로세서인가를 조사하고, 4개 모두 유휴 프로세서가 아닌 경우, 마찬가지로 각각의 서브큐브 0101**, 10**, 11**내의 4개의 프로세서들이 유휴인가를 조사하는 것을 말한다. 만일 10**에 속한 4개의 프로세서, 1000, 1001, 1011, 1010들이 모두 유휴인 경우 프로세서 할당 알고리듬은 이 4개의 프로세서의 할당비트를 1로 만들고, 2차원 서브큐브를 필요로하는 작업에 이 프로세서들을 할당한다.

표 1. $PGG^2{}_4$ 그리고 그에 속하는 $PG^2{}_4$와 GC
Table 1. $PG^2{}_4$ and GC contained in $PGG^2{}_4$.

Gray code	$PG^2{}_4$	$PGG^2{}_4$	Subcube
[1, 2, 3, 4] [1, 2, 4, 3] [2, 1, 3, 4] [2, 1, 4, 3]	[1, 2, *, *] [2, 1, *, *]	bb**	00** 01** 11** 10**
[1, 3, 2, 4] [1, 4, 2, 3] [2, 3, 1, 4] [2, 4, 1, 3]	[1, *, 2, *] [2, *, 1, *]	b*b*	0*0* 0*1* 1*1* 1*0*
[1, 3, 4, 2] [1, 4, 3, 2] [2, 3, 4, 1] [2, 4, 3, 1]	[1, *, *, 2] [2, *, *, 1]	b**b	0**0 0**1 1**1 1**0
[3, 1, 2, 4] [4, 1, 2, 3] [3, 2, 1, 4] [4, 2, 1, 3]	[*, 1, 2, *] [*, 2, 1, *]	*bb*	*00* *01* *11* *10*
[3, 1, 4, 2] [4, 1, 3, 2] [3, 2, 4, 1] [4, 2, 3, 1]	[*, 1, *, 2] [*, 2, *, 1]	*b*b	*0*0 *0*1 *1*1 *1*0
[3, 4, 1, 2] [4, 3, 1, 2] [3, 4, 2, 1] [4, 3, 2, 1]	[*, *, 1, 2] [*, *, 2, 1]	**bb	**00 **01 **11 **10

[정리 1] n차원 하이퍼큐브에서 모든 k차원 서브큐브의 검색을 위해 필요한 $PGG^k{}_n$의 갯수는 C(n, k) 이다.

증명 : 하나의 $PGG^k{}_n$는 (n-k)!개의 $PG^k{}_n$를 포함하며, 다시 각각의 $PG^k{}_n$는 k!개의 그레이코드를 포함한다. 따라서 하나의 $PGG^k{}_n$는 (n-k)!k!개의 그레이코드를 포함한다. 서브큐브의 검색에 서로 다른 C(n, k)개의 $PGG^k{}_n$을 사용하는 경우, 모두 C(n, k)(n-k)!k! = n!개의 그레이코드를 사용한 것과 동일한 효과를 얻는다. 이것은 n비트로 나타낼 수 있는 모든 그레이코드의 수이고, 이것을 사용하면 모든 서브큐브를 검색할 수 있다. Q.E.D

[정리 2] C(n, k)개의 서로 다른 $PGG^k{}_n$를 사용하면 n차원 하이퍼큐브내에 존재하는 모든 C(n, k) 2^{n-k} 개의 서브큐브 번지를 얻을 수 있다.

증명 : 하나의 PGG^k_n로 부터 얻어지는 서브큐브 번지의 갯수는 2^{n-k}이다. 각 PGG^k_n는 서로 다른 위치에 *를 포함하며, 각 PGG^k_n로부터 얻어지는 서브큐브 번지는 서로 다르다. 따라서 n차원 하이퍼큐브에서 가능한 C(n, k)개의 서로 다른 PGG^k_n로 얻어지는 서브큐브 번지의 수는 C(n, k) 2^{n-k}가 된다. 이것은 n차원 하이퍼큐브내에 존재하는 서브큐브의 수와 일치한다. Q.E.D

Chen과 Shin의 다중 그레이코드 방법[9]에서 모든 k차원 서브큐브를 검색하기 위해서는, 차원 k에 관계없이 항상 C(n, n/2)개의 그레이코드를 사용하여야 한다. 따라서 사용되는 서브큐브 번지의 갯수는 C(n, n/2) 2^{n-k+1}이다. 그러나 본 논문에서 제안한 PGGkn를 이용하면 정의 1과 2로부터 모든 k차원 서브큐브의 검색에 필요한 최소의 갯수인 C(n, k)개의 그레이코드를 얻을 수 있다. 따라서 PGG^k_n는 모든 k차원 서브큐브를 찾는데 필요한 최소의 검색범위를 제공한다. 다중 그레이코드에서 사용하는 C(n, n/2)개의 그레이코드는 PGG^k_n에서 k= n/2 인 경우에 해당하며, 이것은 PGG^k_n에서 사용하는 그레이코드 갯수의 상한치이다. 그러나, 할당비트 참조 횟수에서는 k= n/2 인 경우에도 PGG가 다중 그레이코드 방법의 1/2이 된다. 다음은 PGG^k_n을 이용한 프로세서 할당과 할당해제 과정을 나타낸 것이다.

프로세서 할당

Step 1 : 작업에 필요한 서브큐브의 차원, k를 결정한다.

Step 2 : C(n, k)개의 PGG^k_n을 생성하여, 각 PGG^k_n로부터 얻어지는 서브큐브 번지를 사용하여 차례로 k차원 서브큐브를 검색한다. 만일 k차원 서브큐브를 찾는데 실패하면 step 4로 분기한다.

Step 3 : step 2에서 찾아낸 k차원 서브큐브에 작업을 할당하고, 그 서브큐브에 속하는 노드들의 할당비트를 1로 만들고 프로세서 할당을 마친다.

Step 4 : 작업을 큐(queue)에 대기시킨다.

프로세서 할당해제

할당해제 되는 서브큐브에 속하는 모든 프로세서들의 할당비트를 0으로 만든다.

Ⅳ. 시뮬레이션 및 성능 비교

프리 리스트(FL)방법의 경우는 매우 복잡한 할당해제 과정이 필요하다. 프로세서의 할당해제시마다 유휴인 서브큐브의 목록을 갱신해야 하는데, 목록의 길이를 O(n)이라고 했을 경우 총 시간 복잡도는 O(n3)이 된다.[11] 만일 0 또는 1차원 서브큐브 목록으로만 구성된 경우, 목록의 길이는 $O(n^3)$까지 커질 수 있으며, 총 시간 복잡도는 $O(n^2 2^{n-k})$까지 증가한다.[10]

트리병합(TC)방법에서는 C-변환을 사용하는 경우 하이퍼큐브를 트리로 나타내어야 하며, 트리의 링크를 재배열하는 트리병합 과정이 필요하다. C-변환을 간단히 한 R-변환을 사용하는 경우는 검색집합(search set:압축 그레이코드 그룹과 같은 역할)의 결정을 위해 재귀호출을 이용한 알고리듬을 사용해야하는 단점이 있다. 486DX2-50 CPU를 장착한 시스템상에서 C언어를 사용하여 TC방법에서 검색집합을 생성하는 R-변환과 PGG방법에서 PGG를 생성하는 알고리듬을 비교하였다. 15차원 하이퍼큐브에서 7차원 서브큐브의 검색을 1000회 수행하는 동안 R-변환과 PGG를 생성하는데 걸리는 시간을 비교한 결과, R-변환은 16.81초 PGG방법은 11.04초가 소요되었다. 이 차이는 TC방법에서 검색집합의 생성을 위해 재귀적 알고리듬을 사용해야하기 때문에 발생한다. 큐브병합(CC)방법 역시 재귀 알고리듬을 사용하고 있다. 일반적으로 검색집합 생성에서 재귀 알고리듬을 사용하면 그 실행시간이 길어지고, 또한 프로그래밍 기술에 따라 실행시간은 차이가 있을 수 있다. 그리고 TC방법의 R-변환을 사용하면 유휴 서브큐브를 찾기위한 서브큐브 번지의 생성순서가 항상 일정하여 검색순서가 고정되어 있는 반면, PGG방법에서는 검색에 사용되는 PGG의 순서를 바꿈으로서 융통성있는 검색순서를 제공한다.

프로세서 할당방법의 성능은 프로세서의 이용도로 비교할 수 있다. 하이퍼큐브 상에서 프로세서의 이용도라고 하는 것은 평균 작업 중인 프로세서의 갯수와 총 프로세서의 비이다. 다음은 프로세서의 이용도(utilization ratio)를 구하는 식이다.

$$u = \frac{\sum_{i=1}^{r} s_i t_i}{NT}$$

s_i : i 번째 작업에 필요한 프로세서의 수
t_i : i 번째 작업 수행에 걸리는 시간
N : 하이퍼큐브내의 총 프로세서의 수
T : 전체 시간
r : 전체 작업수

이와 같은 프로세서 이용도를 높이기 위해서는 프로세서 할당방법이 짧은 시간내에 작업이 요구하는 크기의 유휴 서브큐브를 찾아낼 수 있는 능력을 갖고 있어야 한다.

표 2는 BD, GC, TC, CC, PGG방법들의 검색 가능한 서브큐브의 갯수를 나타낸 것이다. TC나 PGG방법의 경우 존재하는 모든 서브큐브를 검색할 수 있는 반면 할당비트의 참조 횟수는 BD나 GC방법보다 증가하게 된다.

표 2. n차원 하이퍼큐브에서 각 알고리듬이 검색 가능한 k차원 서브큐브의 수

Table 2. Number of k-D subcubes that each algorithm can search in n-D hypercube.

	0-D	k-D (1≤k≤n-1)	n-D
Total subcube #	2^n	$C(n, k)2^{n-k}$	1
BD	2^n	2^{n-k}	1
GC	2^n	$2(2^{n-k})$	1
TC, CC, PGG	2^n	$C(n, k)2^{n-k}$	1

표 3. n차원 하이퍼큐브에서 k차원 서브큐브 검색시 각 방법이 최악의 경우 참조하게 되는 할당비트의 수

Table 3. Number of allocation bits accessed to search a k-D subcube in an n-D hypercube in the worst case.

할당 알고리듬	0-D	k-D (1≤k≤n-1)	n-D
BD	2^n	2^n	2^n
GC	2^n	$2(2^n)$	2^n
Multiple GC	$C(n, \lfloor n/2 \rfloor)2^{n-1}$	$C(n, \lfloor n/2 \rfloor)2^{n-1}$	$C(n, \lfloor n/2 \rfloor)2^{n-1}$
TC, PGG	2^n	$C(n, k)2^n$	2^n

표 4. 시뮬레이션에서 발생되는 작업의 분포

Table 4. Distribution of tasks generated in the simulaution.

작업발생간격	T_i를 평균값으로 갖는 지수분포 (T_i = 평균 작업수행시간 × $E(s_i)$ × # / 2^n)	
작업수행시간	[10, 20] 사이의 균일 분포	
서브큐브의 차원	균일분포	p(k) = 1/(n+1)
	정규분포	p(k) = C(n, k) × 0.5^n
	지수분포	p(k) = 0.3 × 0.7^k

표 3은 최악의 경우 각 방법이 참조하게 되는 할당비트의 수를 나타낸 것이다. 하나의 할당비트의 참조에 걸리는 시간은 메모리를 한번 참조하는데 걸리는 시간이므로 매우 짧은 시간이다. 그러나 이용가능한 k차원 서브큐브가 없는 경우에는 모든 k차원 서브큐브를 검색하게 되며, 이러한 최악의 경우 할당비트의 참조횟수는 $C(n, k)\ 2^n = (n!/(n-k)!k!)2^n$ 으로, 결국은 실패할 프로세서 할당의 경우에 제일 긴 시간을 소모하게 된다.

비록 TC나 PGG방법이 모든 서브큐브를 검색할 수 있더라도 서브큐브를 찾아내기 위해 많은 시간이 걸린다면 오히려 프로세서의 이용도를 감소시키게 된다. 그림 4는 10차원 하이퍼큐브에서 k차원 서브큐브 검색시 다중 그레이코드 방법과 TC, PGG 방법이 최악의 경우 참조하게 되는 할당비트의 수를 그래프로 나타낸 것이다. k = 5인 경우 다중 그레이코드 방법과 PGG 방법은 같은 수의 그레이코드를 사용하지만, 할당비트 참조 횟수에서는 PGG가 다중 그레이코드 방법의 1/2이 됨을 보여주고 있다.

표 4는 시뮬레이션에서 사용되는 작업의 확률적 분포이다. 입력된 작업이 할당된 서브큐브에서 수행되는데 걸리는 시간은 [10, 20]사이의 균일분포(uniform distribution)를 갖는 것으로 가정하였다. 입력되는 작업의 발생간격은 다음식으로 부터 결정하였다.

$$E(s_i)E(N) \le 2^n$$

s_i : i번째 작업에 필요한 프로세서의 수
$E(s_i)$: 작업에 필요한 프로세서 수의 평균값
E(N) : 평균 작업수행시간 동안 발생되는 작업의 수
2^n : n차원 하이퍼큐브의 프로세서 수

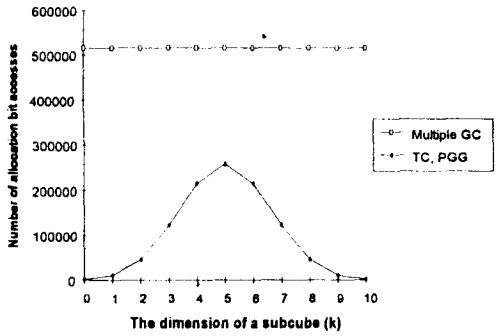
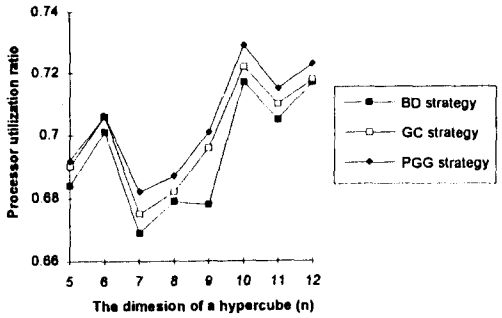


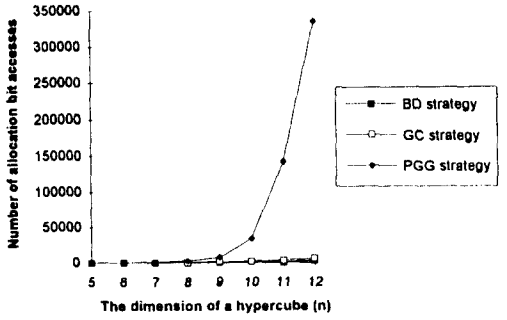
그림 4. 10차원 하이퍼큐브에서 k차원 서브큐브 검색시 다중 그레이코드방법, TC, 그리고 PGG 방법이 최악의 경우 참조하게 되는 할당비트의 수

Fig. 4. Number of allocation bits accessed by Multiple GC, TC, and PGG to search a k-D subcube in a 10-D hypercube in the worst case.

위 식은 작업을 수행중인 서브큐브들의 평균 노드갯수가 n차원 하이퍼큐브의 노드 갯수인 2n을 넘지 않게 하기 위한 평균 작업발생간격을 계산하는 식이다. 다시 E(si)와 E(N)은 각각 다음 식으로 부터 얻어지며, 여기서 P(k)는 작업이 k차원 서브큐브를 필요로 할 확률이다.



(a)



(b)

그림 5. (a) 프로세서 이용도.
(b) 할당비트의 참조횟수

Fig. 5. (a) Processor utilization ratio.
(b) Number of accesses to allocation bits.

$$E(s_i)=\sum_{k=0}^{n} P(k)2^k$$

E(N)=평균 작업발생율 × 평균 작업수행시간 = $\frac{\text{평균 작업수행시간}}{\text{평균 작업발생간격}}$

여기서 E(N)을 위식에 대입하면

$$E(s_i) \times \frac{\text{평균 작업수행시간}}{\text{평균 작업발생간격}} \le 2^n$$

와 같이된다. 평균 작업발생간격에 대하여 정리하면 다음 식을 얻는다. 여기서 w(0≤w≤1)는 평균 작업발생간격을 가변시키기 위한 변수이다.

$$\text{평균 작업발생간격} = \text{평균 작업수행시간} \times \frac{E(s_i)}{2^n} w$$

표 4로부터 작업수행시간은 [10, 20]사이의 균일분포이므로 평균 작업수행시간은 15가 되고, $E(s_i)$의 값은 작업에 필요한 서브큐브 차원의 분포에 따라 결정되며, w값은 0.9를 사용하여 평균 작업발생간격을 얻었다.

본 논문은 BD, GC, PGG방법에 관하여 시뮬레이션을 수행하였다. TC방법의 경우는 PGG방법과 비교하여 실제 알고리듬 수행시간의 차이를 제외하고는 같은 결과를 얻으므로 생략하였다. 그림 5의 (a)와 (b)는 요구되는 서브큐브 크기가 정규분포인 경우에 대해 시뮬레이션으로 부터 얻은 BD, GC, PGG방법의 프로세서 이용도와 할당비트 참조횟수를 각각 나타낸 것이다. 프로세서 이용도에 있어서는 PGG방법이 BD방법이나 GC방법보다 우수함을 알 수 있다. 이것은 프로세서 할당 알고리듬이 검색할 수 있는 서브큐브의 수가 많을수록 프로세서 이용도가 커짐을 나타낸다. 그러나 할당비트의 참조횟수를 보면 PGG방법이 다른 두 방법보다 많음을 알 수 있다. 이것은 검색가능한 서브큐브의 수가 많은 만큼 더 많은 할당비트를 참조해야 하는 데 원인이 있다. 하이퍼큐브의 차원이 증가하여도 프로세서의 이용도는 세 방법 모두에서 큰 차이를 보이지 않으나, 할당비트의 참조횟수는 PGG방법의 경우 현저히 증가한다. 이것은 하이퍼큐브의 차원이 증가하면 가능한 서브큐브의 갯수는 지수적으로 증가하며 PGG방법은 이러한 서브큐브를 모두 검색하기 때문이다.

그림 6은 하나의 작업이 작업대기 큐에서 기다리는 대기시간의 평균값을 나타낸 것이다. 시뮬레이션의 단위시간을 0.1초로 가정하면, 대기시간의 차가 비교적 적은 10차원의 경우 GC방법과 PGG방법의 평균 작업대기시간은 각각 496ms, 465ms 이다. 두 방법 사이에는 31ms의 차이가 있다. 한편 할당비트 참조횟수를 보면 GC방법이 1365, PGG방법이 34241로 32876의 차이가 있다. 이 차이에 하나의 할당비트 참조에 걸리는 시간을 곱하면 전체 참조에 걸리는 시간을 얻을 수 있다. 한 워드, 즉 32비트를 참조하는데 100ns가 걸린다면 한 비트에 (100/32)ns가 걸린다고 볼 수 있고, GC방법과 PGG방법간의 할당비트 참조에 걸리는 시간차는 약 0.1ms가 된다.

이것은 PGG방법의 경우 할당비트 참조 횟수가 비록 많더라도 성능향상에 큰 결점이 되지 않는다는 것을 나타낸다.

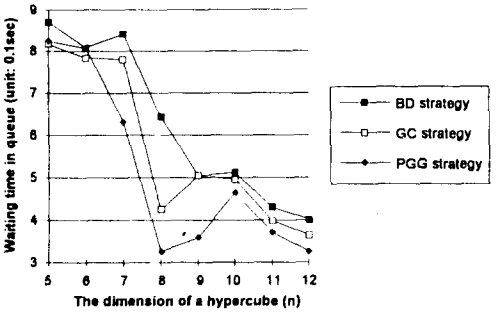
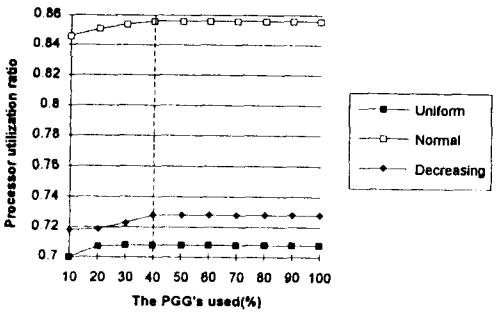
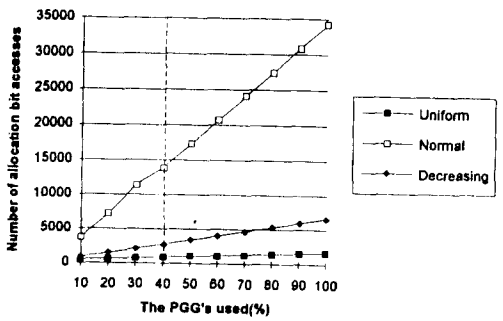


그림 6. 작업의 큐 대기시간
Fig. 6. Task waiting time in a queue.



(a)



(b)

그림 7. (a) 프로세서 이용도.
(b) 할당비트의 참조횟수
Fig. 7. (a) Processor utilization ratio.
(b) Number of accesses to allocation bits.

n차원 하이퍼큐브 내에 존재하는 k차원 서브큐브를 모두 검색하기 위해서는 C(n, k)개의 PGG^k_n가 필요함을 이미 앞에서 보였다. 사용되는 PGG^k_n을 증가시키면서 이용도의 증가를 시뮬레이션을 통하여 알아 보았다. 그림 7은 10차원 하이퍼큐브에서 PGG^k_{10}의 증가에 따른 이용도 및 할당비트의 참조횟수의 변화를 보여주고 있다. 그림에서 x축은 사용한 PGG^k_{10}의 수를 C(10, k)를 100으로한 퍼센트 값으로 나타낸 것이며, y축은 각 경우의 프로세서 이용도를 나타낸 것이다. 우선 서브큐브의 크기가 균일분포인 경우를 보면 30%까지는 PGG의 수를 증가시키면 프로세서의 이용도는 0.7에서 0.708로 증가한다. 그러나 30%이상을 넘게되면, 아무리 PGG^k_{10}의 수를 증가시켜도 프로세서의 이용도는 0.708이상 증가하지 않는다. 서브큐브의 크기가 정규분포인 경우 40%가 되면 프로세서의 이용도는 0.856로 더이상 증가하지 않는다. 지수적으로 감소하는 분포에서도 역시 40%가 포화값이 되어 프로세서의 이용도는 0.728 이상 증가하지 않는다.

위 시뮬레이션 결과로부터 n차원 하이퍼큐브에서 k차원 서브큐브의 검색을 위해 필요한 C(n, k)개의 PGG 중 40%만 사용해도 모든 PGG를 사용한 경우의 프로세서 이용도와 같은 결과를 얻게 된다. 즉 40%의 PGG만 사용하게 되면 비록 모든 서브큐브의 검색은 불가능하지만 프로세서 이용도에 있어서는 100% 모두 사용한 것과 같으며, 또한 할당비트의 참조횟수도 적어지게 된다. 사용중인 서브큐브가 하이퍼큐브 내에 불규칙적으로 분포하고 있으므로 어떤 종류의 PGG를 사용하여도 동일한 프로세서 이용도를 얻으며, PGG 생성 또한 자유롭고 간단하기 때문에, 임의의 40%의 PGG를 선택하는 과정에서의 추가 부담은 발생하지 않는다.

작업이 할당된 서브큐브가 많은 경우 검색하여야 하는 서브큐브의 갯수는 급격히 증가한다. 예를 들어 유휴상태인 n차원 하이퍼큐브에서 (n-1)차원 서브큐브를 처음 할당하는 경우 첫번째 시도에서 성공하게 된다. 그러나 두번째 (n-1)차원 서브큐브를 할당하는 경우 n차원 하이퍼큐브에서 가능한 2n개의 (n-1)차원 서브큐브 중 1개 만이 할당가능한 서브큐브를 이루고 있다. 따라서 나머지 하나를 할당하기 위하여 최악의 경우 (2n-1)개의 서브큐브를 검색하여야 한다. 일반적으로 처음 몇번 시도에서 프로세서 할당에 실패하는 경우, 이용 가능한 서브큐브가 거의 없다는 것을 의미하며 시도 횟수를 증가시켜도 할당에 성공할 확률은 상당히 낮다.

V. 결 론

MIMD 하이퍼큐브 시스템에서 효과적인 프로세서 할당은 시스템 자원의 이용도를 높이기 위한 중요한

작업이다. 본 논문에서는 하이퍼큐브를 위한 여러가지 할당 방법들을 살펴보고, 기존 방법의 문제점을 보완한 새로운 PGG방법을 제안하였다. 본 논문에서 제안한 PGG방법은 서브큐브의 검색에 압축 그레이 코드 그룹을 사용하는 할당 알고리듬으로, 하이퍼큐브상에서 모든 서브큐브를 찾는 문제에 대하여 간단하면서도 효율적인 해답을 제시한다. n차원 하이퍼큐브에서 PGG방법을 사용하여 서브큐브를 검색하는 경우, C(n, k)개의 PGG^k_n를 사용하면 모든 k차원 서브큐브를 찾을 수 있다.

PGG방법은 동일한 검색범위를 갖는 TC방법과 할당작업의 수행시간을 비교한 결과, 재귀적 알고리듬을 사용하는 TC방법에 비해 짧은 알고리듬 수행시간을 보였다. 또한 시뮬레이션을 통하여 BD방법, GC방법과 비교한 결과 프로세서의 이용도에 있어서 가장 높은 값을 나타내었다. 이러한 결과는 검색범위에 있어서 PGG방법이 BD방법이나 GC방법보다 많은 서브큐브를 검색할 수 있는 능력을 갖추었기 때문이었다.

그러나 검색범위가 커짐으로해서 할당 비트의 참조 횟수는 다른 방법보다 많으며, 이와 같이 많은 할당 비트의 참조 횟수를 줄이기 위해 서브큐브 검색에 사용된 PGG의 수를 적게한 결과, C(n, k)개의 PGG 중 40%정도만 사용하더라도 C(n, k)개 모두를 사용한 것과 같은 이용도를 얻음을 본 논문의 시뮬레이션을 통하여 발견했다. 다시 말하면 처음 몇번 시도에서 프로세서 할당에 실패하는 경우 이용 가능한 서브큐브가 거의 없으며, 따라서 40% 이상의 PGG를 사용해도 할당에 성공할 확률은 거의 없다는 것을 의미한다. 이러한 결과는 다른 프로세서 할당방법에도 그대로 적용될 수 있으며, 전체검색 범위의 40%가 프로세서 이용도를 높이는 한계치가 됨을 의미한다.

參 考 文 獻

[1] Y. Saad and M. M Schutz., "Topologies properties of hypercubes." *IEEE Trans. Comput.*, vol. C-37, pp.867-872, July 1988.

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

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

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

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

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

[7] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw- Hill, pp. 32-35, 1985.

[8] 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.

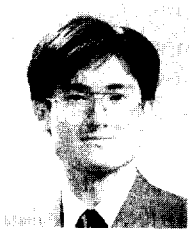
[9] M. S. Chen and K. G. Shin, "Processor Allocation in an N-Cube Multiprocessor Using Gray Codes," *IEEE Trans. Comput.*, vol. C-36, pp. 1396-1407, Dec. 1987.

[10] P. J. Chuang and N. F. Tzeng, "A Fast Recognition Complete Processor Allocation Strategy for Hypercube Computers." *IEEE Trans. Comput.*, vol. 41, pp. 467-479, Apr. 1992.

[11] J. Kim, C. R. Das and W. Lin, "A Top-Down Processor Allocation Scheme for Hypercube Computers," *IEEE Trans. Comput.*, vol. 2, pp. 20-40, Jan. 1991.

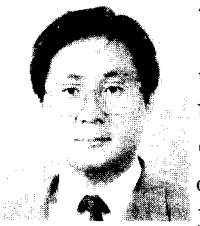
[12] G. Kim and H. Yoon, "A processor allocation strategy using cube coalesing in hypercube multicomputers." *Proc. of the 5th IEEE Symp. on Parallel and Distributed Processing*, pp. 596-605, Dec. 1993.

著者紹介



李 昇 勳(準會員)

1969年 1月 8日生. 1992年 인하대학교 전자공학과 졸업(학사). 1994年 인하대학교 전자공학과 졸업(석사). 1994年 ~ 현재 에스티 엠(주) 근무



崔 相 昉(正會員)

1954年 9月 11日生. 1981年 한양대학교 전자공학과 졸업. 1988年 University of Washington 졸업(공학 석사). 1990年 University of Washington 졸업(공학 박사). 1981年 ~ 1986年 금성정보통신(주) 근무 1991年 ~ 현재 인하대학교 전자공학과 조교수. 주관심 분야는 컴퓨터 구조, 병렬 및 분산처리 시스템, Fault-tolerant computing 등임.