# A Fast Algorithm For Complete Subcube Recognition

Hal J. Burch *

Computer Science Department
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3891
hburch+@cs.cmu.edu
http://www.cs.cmu.edu/~hburch/

Fikret Ercal

Computer Science Department and
Intelligent Systems Center
University of Missouri-Rolla
Rolla, MO 65401
ercal@cs.umr.edu
http://www.cs.umr.edu/~ercal/

**Abstract**

*The complete subcube recognition problem is defined as, given a collection of available processors on an n-dimensional hypercube, locate a subcube of dimension k that consists entirely of available processors, if one exists. Despite many algorithms proposed so far on this subject, improving the time complexity of this problem remains a challenge. Efficiency limits that can be reached have not been exhausted yet.*

*This paper proposes a novel algorithm to recognize all the overlapping subcubes available on an n-dimensional hypercube whose processors are partially allocated. Given $P = 2^n$, as the total number of processors in the hypercube, the new algorithm runs in $O(n \cdot 3^n)$ or $O(P^{\log_2 3} \log_2 P)$ time which is an improvement over previously proposed strategies, such as multiple-graycode, missing combination, maximal set of subcubes, and tree collapsing.*

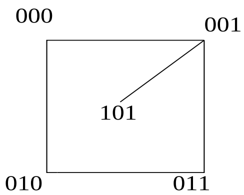
## 1 Introduction

Although interest has shifted to other topologies in recent years, the hypercube remains an important and well-studied topology in parallel computing. One of the interesting problems with the hypercube is the allocation of subcubes within a hypercube. That is, given a collection of available processors on a n-dimensional hypercube and a dimension k, allocate (or fail to allocate) a subcube of dimension k, where each processor within that subcube has not been allocated yet. An algorithm for allocation is said to have complete subcube recognition ability if and only if the allocation fails only when there is no available subcube of the requested dimension.

There are several algorithms that exist that have the complete subcube recognition property, such as the multiple-graycode (multiple-GC) [1], the maximal set of subcubes [2], tree collapsing (TC) [3], and missing combination (MC) [4]. Parallel complete subcube recognition algorithms are also proposed [5]. The fastest of these algorithm run in approximately $O(2^n \cdot \binom{n}{j})$, which for $j = \frac{n}{2}$ is worse than $O(\frac{4^n}{n}) = O(\frac{P^2}{\log_2 P})$, where $P = 2^n$, the number of processors in an n-dimensional hypercube. This paper describes a new algorithm, *subcube building*, to determine all the subcubes (possibly overlapping) that exists consisting entirely of available processors, and some adaptations of the scheme to the dynamic allocation problem. In addition, it will be shown that the time complexity of the subcube building algorithm is $O(n \cdot 3^n)$ or $O(P^{log_2 3} \cdot \log_2 P)$. For the case of searching for a subcube of a given dimension k, this algorithm will be shown to be able to be slightly improved to $O(n \cdot \sum_{j=0}^{k} \binom{n-k+j}{j} 2^{n-j})$.

All of the allocation schemes mentioned above are known to be *statically optimal*, as the allocation methods described in this paper. A scheme is said to be statically optimal if for any sequence of allocation requests such that the total number of processors requested is less than the number of processors in the hypercube, the algorithm will satisfy all requests. Several other allocation schemes have been proposed that, while statically optimal, do not have complete subcube recognition. These include the buddy and graycode [1] strategies. These have time complexities of $O(2^n)$, but recognize $1/\binom{n}{k}$ and $(n-k+1)/\binom{n}{k}$ of the possible subcubes respectively. Thus, they are not examined in this paper, and neither the fast maximum set of subcubes [2], which is a heuristic algorithm to approximate the maximum set of subcubes.

*This represents work done at the University of Missouri - Rolla




| Start: | 1st pass | 2nd pass |
|---|---|---|
| 000 | 00* | 0** |
| 001 | 01* | |
| 010 | 0*0 | |
| 011 | 0*1 | |
| 101 | *01 | |

Figure 1: Example of finding available subcubes in a 3-d hypercube

## 2 Subcube Building Algorithm

Subcubes of a n-dimensional hypercube can be represented as a n-tuple of $0, 1, *$, where $*$ signifies "don't care." For example, $0 * 1 *$ contains 0010, 0011, 0110, and 0111. The dimension of a subcube is exactly the number of *'s in its representation. With this representation, one can clearly see that the total number of different subcubes in all dimensions (not necessarily disjoint) is equal to $3^n$.

The subcube building algorithm presented here determines all available subcubes of any dimension that exist within an n dimensional hypercube. The idea of the algorithm is to start with all the 0 dimensional available subcubes (which are, of course, just individual processors that are not yet allocated). Then, try to join them into larger subcubes, and add the resulting subcubes into the available list and iterate this process until no more subcubes are found.

The joining process is fairly simple. The idea is to repetitively join two subcubes whose representation differ only in one position. If two subcubes are available which differ on in the d'th position (and neither of them have a "don't care" in that position), then the subcube with the same representation with a "don't care" in the d'th position is also available, so it can be added to the list of available subcubes. For example, 001*0 and 011*0 can be joined to create 0*1*0.

As an example, let's examine Figure 1. Here, the beginning set would be 000, 001, 010, 011, and 101. 000 and 001 are joined to create the subcube 00*, 010 and 011 are joined to create 01*, and so on, to create 00*, 01*, 0*0, 0*1, and *01. 00* and 01* can be joined to create 0**, giving the only two dimensional subcube available. This obviously cannot be joined to create a 3 dimensional subcube, so the available subcubes are 000, 001, 010, 011, 011, 00*, 01*, 0*0, 0*1, *01, and 0**.

The implementation chosen utilizes a $3^n$ array of boolean values, where each position represents the availability of a subcube. Let's represent the $i^{th}$ element of this array with Q[i]. Let $d(M)$ represent the dimensionality of subcube M and $m_i$ be the $i^{th}$ element in its n-tuple representation. A total ordering is defined on the subcubes as follows: Consider two subcubes A and B. $A < B$ if $d(A) < d(B)$, or if $d(A) = d(B)$ and $a' < b'$, where $a'$ is formed by replacing "*"s in $a$ with "1"s and other bits (0 or 1) with "0"s ($b'$ is formed in a similar way). Additionally, if $d(A) = d(B)$ and $a' = b'$, then $A < B$ if $a'' < b''$, where $a''$ is obtained by replacing "*"s in $a$ with "1"s and leaving the other bits intact ($b''$ is formed in a similar way). Basically, the ordering is based on dimension first, then location of don't cares, and lastly the value of the other bits. For example, $0*0** > 01*** > 101** > 100**$.

Using this ordering, the position of the position of a subcube in the Q[] array is somewhat difficult to determine. The first subcube of dimension d is at

$$\sum_{j=0}^{d-1} \binom{n}{j} 2^{n-j}$$

This is fairly simple to see, since there are $\binom{n}{j}$ ways to select j "don't care" positions in a subcube representation, and $2^{n-j}$ subcubes with those positions set to "don't care" ($n-j$ positions, 2 values). Then, for each subcube below it in the ordering, add $2^{n-d}$. If the bit positions are $b_0, b_1, ..., b_{d-1}$, ordered from highest bit to lowest (the lowest valued bit being bit 0) , then this is an additional

$$\sum_{j=0}^{d-1} \binom{b_j}{d-j} 2^{n-d}$$

Then, take the representation of the subcube dropping the positions with "don't care" and add that interpreted as a binary number. This takes $O(n \cdot d)$ time for each of the first two steps, and $O(n)$ for the last step, for a total of $O(n \cdot d)$.

However, an auxiliary array of size $2^n$ can be maintained that specifies the starting location of the collection of subcubes with the bits set to "don't care", and also the dimension of a subcube with those bits set. This array can be built in $O(n2^n)$ time by starting the position at 0, looping through each dimension of subcube, and setting the start of the block appropriately. Algorithm 1 shows how to setup this auxiliary array. With this auxiliary array, determining the position of a subcube is a $O(n)$ operation.

Once this array has been built, the subcube building algorithm is fairly simple. Loop through all the dimension of the subcubes, from 1 to n. For each subcube of the given dimension, find the first position

*Algorithm 1 :* Setting up auxiliary array

```
/* S = auxiliary array of start of blocks of subcubes
   n = dimension of the overall hypercube
   d = dimension of subcubes within a block      */
begin
   count = 0
   for each dimension of subcube d (from 1 to n)
      for x = each number < 2^n with d 1's
         S_x = count
         count = count + 2^(n-d)
      endfor
   endfor
end
```

*Algorithm 2 :* Subcube_Building

```
/* n = dimension of the overall hypercube
   P = boolean array of availability of the processors
   Q = 3^n boolean array, all initialized to 0
   x = don't care bits in subcube attempted to be built
   y = don't care bits in subcubes being combined
   i = value of other bits in subcubes being combined */
begin
   for each processor i,
      Q_i = P_i     /* initialize */
   endfor
   /* build all available subcubes */
   for each dimension d (d = 0 to n-1)
      for x = each number < 2^n with d 1's
         position = S_x
         b = highest position with a 1 in x
         y = x with b set to 0
         for i = 0 to 2^(n-d+1) - 1
         /* The subcube being considered has *'s
             wherever x has a 1 and the other
             bits are defined by i       */
            if the (b - d + 1)-th bit of i is FALSE
               Q_position = Q_(i+S_y) AND Q_(i+2^(b-d+1)+S_y)
               position = position + 1
            endif
         endfor
      endfor
   endfor
end
```

which is a "don't care," and check to see if the subcube with 0 in that position and the subcube with 1 in that position are both available. If they are, then this subcube is available, otherwise, it is not.

Figure 2 shows how the algorithm 2 works on a 3 dimensional hypercube. The calculation of x, b, and y

| | | |
|---|---|---|
| x=0 | 000 001 010 011 100 101 110 111 | S[0] = 0 |
| x=1 | 00* 01* 10* 11* | S[1] = 8 |
| x=2 | 0*0 0*1 1*0 1*1 | S[2] = 12 |
| x=3 | 0** 1** | S[3] = 20 |
| x=4 | *00 *01 *10 *11 | S[4] = 16 |
| x=5 | *0* *1* | S[5] = 22 |
| x=6 | **0 **1 | S[6] = 24 |
| x=7 | *** | S[7] = 26 |

| | |
|---|---|
| Q[8] = Q[0] AND Q[1] | Q[9] = Q[2] AND Q[3] |
| Q[10] = Q[4] AND Q[5] | Q[11] = Q[6] AND Q[7] |
| Q[12] = Q[0] AND Q[2] | Q[13] = Q[1] AND Q[3] |
| Q[14] = Q[4] AND Q[6] | Q[15] = Q[5] AND Q[7] |
| Q[16] = Q[0] AND Q[4] | Q[17] = Q[1] AND Q[5] |
| Q[18] = Q[2] AND Q[6] | Q[19] = Q[3] AND Q[7] |
| Q[20] = Q[8] AND Q[9] | Q[21] = Q[10] AND Q[11] |
| Q[22] = Q[8] AND Q[10] | Q[23] = Q[9] AND Q[11] |
| Q[24] = Q[12] AND Q[14] | Q[24] = Q[13] AND Q[15] |
| Q[26] = Q[20] AND Q[21] | |

Figure 2: Illustration of algorithm 2 on a complete 3-d hypercube

are excluded, but the method of determination of $Q_i$ is noted, in the proper order.

Figure 3 shows the values of Q after the algorithm is complete. Note that $Q[8]$ represents 00*, $Q[9]$ represents 01*, $Q[12]$ 0*0, $Q[13]$ 0*1, $Q[17]$ *01, and $Q[20]$ 0**, which are exactly the higher dimension subcubes that are shown to be available in Figure 1.

## 3 Complexity Analysis

Basically, this algorithm consists of two steps: the determination of x, b, and y, and the looping through i in algorithm 2. The determination of x can be done fairly easily by looping through all $x < 2^n$ and counting 1's. Each pass takes $O(n2^n)$ time, and a total of n passes are made, for a total of $O(n^2 2^n)$. The determination of b is $O(n)$ operation, and is done $2^n$ times, for a total of $O(n2^n)$. y's calculation is $O(n)$ operation (left shift of 1 and a XOR), and is done $2^n$ times, for a total of $O(n2^n)$. Thus, the x, b, and y determination takes a total time of

$$O(n^2 2^n) + O(n2^n) + O(n2^n) = O(n^2 2^n)$$

Each iteration of the i loop takes $O(n)$ time, to determine the bit of i, do the addition, and array lookups. First, notice that each subcube's representation is a ternary array of length n. This means that there are a total of $3^n$ possible subcubes of a hypercube of dimension n. Note that $Q_j$ is altered exactly once, when $i = j - S_x$, for the x such that $S_x \leq j < S_{x+1}$. Therefore, each subcube is checked exactly once, in $O(n)$ time, giving this portion of the algorithm a time complexity of $O(n \cdot 3^n)$. Thus, the

Q initialized to {1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
First pass: Q[8] = 1, Q[9] = 1, Q[10] = 0, Q[11] = 0, Q[12] = 1, Q[13] = 1,
Q[14] = 0, Q[15] = 0, Q[16] = 0, Q[17] = 1, Q[18] = 0, Q[19] = 0
Second pass: Q[20] = 1, Q[21] = 0, Q[22] = 0, Q[23] = 0, Q[24] = 0, Q[25] = 0
Third pass: Q[26] = 0
Q ends up: {1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0}

Figure 3: Illustration of algorithm 2 on the cube of Figure 1

overall algorithm has a run-time complexity of:

$$O(n^2 2^n) + O(n \cdot 3^n) = O(n \cdot 3^n)$$

Since $P = 2^n$, this gives

$$O(n \cdot 3^n) = O(P^{\log_2 3} \cdot \log_2 P) \approx O(P^{1.585} \cdot \log_2 P)$$

An important consideration is the memory utilization of the algorithm and the time complexity of the setting up of the auxiliary array. The algorithm utilizes a total of $O(3^n)$ memory, using a boolean value for each possible subcube. This is a large amount of storage for high dimension hypercubes. The complexity of setting up the auxiliary array is $O(n2^n)$, which is insignificant with respect to the complexity of the algorithm, so this does not add to the overall time complexity of the algorithm.

If the question is what subcubes of a given dimension are available, an improvement can be made over the $O(n3^n)$ algorithm, but stopping at $d = k$, where $k$ is the requested dimension. This reduces the run-time to $O(n \cdot \sum_{j=0}^{k} \binom{n}{j} 2^{n-j})$ (the $\binom{n}{j}$ is the number of x's that exist with i 1's. The $2^{n-j}$ is the number of i's that exist). One can do even better than this by noticing that the algorithm always adds "*"s to the beginning of the subcube, so if a subcube of dimension m doesn't have $k - m$ bit positions before the first "*"s, it can never be used to build up a k dimensional subcube. Thus, x need only range up to $2^{n-k+d}$. This gives a complexity of $O(n \cdot \sum_{j=0}^{k} \binom{n-k+j}{j} 2^{n-j})$, an improvement for general j, albeit a minor one for the worse case selection of j.

### 3.1 Comparison To Other Techniques

The majority of the algorithms proposed thus far [1]-[5] run in about $O(4^n)$ time, or $O(P^2)$, where $P = 2^n$ is the number of processors in the hypercube. Multiple-GC [1] is $O(C(n, \lfloor \frac{n}{2} \rfloor))2^n$. MC [4] is $O(r2^{n-k}\binom{n}{k})$, where r is the number of allocated subcubes and k is the dimension sought. TC [3] is $O(C(n,k)2^{n-k}\eta(2^k)$, where $\eta(2^k)$ is the time to search a block of size $2^k$. Since $C(n,k)$ is maximum at $k = \frac{n}{2}$, and $lim_{n\to\infty}\frac{C(n,n/2)}{2^n} = 0$, these algorithms all run in about $O(4^n)$ (Note that $lim_{n\to\infty}\frac{C(n,n/2)n}{2^n} = \infty$). The *free-list strategy* [6] has a time complexity that is heavily dependent on the sizes of the maintained free-lists. The complexity given in the paper is from the observed size of the free-lists, $O(n)$, but the complexity of a free-list corresponding to a subcube of dimension k can grow up to $2^{n-k-1}$. Furthermore, the paper does not provide a detailed analysis of the number of pairwise comparisons that need to be performed for merging purposes whenever there is a deallocation. An analysis of the time complexity more rigorous than the one provided in [6] is needed in order to make a fair comparison between the free-list strategy and our technique. Therefore, the free-list strategy is excluded from comparison.

If the first subcube found of the requested dimension is used (the simplest method, "first match"), then in the case where the buddy system detects a subcube, the "first match" approach finds the same subcube. Basically, the buddy system will find the subcubes where the last k positions are "don't care"'s. If we order the search of x's so that the first x it tries is the one where the last d bits are 1, and the rest are 0, then if there exists an available subcube of the requested dimension that the buddy strategy would find, then the subcube building algorithm will match that one first. Therefore, this "first match" strategy is statically optimal.

Alternatively, this algorithm can be adapted to the Bipartite algorithm [4] to create a fast allocation strategy. The bipartite strategy selects the available subcube which resides in the fewest number of available subcubes of highest dimension available. The adaptation goes as follows: Let k be the dimension of the subcube requested. Initialize an array of counts to be 1 for all the available subcubes of maximum dimension. Then, recursively add the count of a subcube to each of the two subcubes that are created by breaking the original subcube along its highest dimension. Once this has been done down to k+1 dimensional cubes, allow the final splitting to be done over any of the remaining "don't cares." The formalization of this is shown as algorithm 3. The initialization of the array takes $O(3^n)$ time, and the breaking down of the cubes takes $O(n \sum i = k+1^d \binom{n}{i} 2^{n-i})$ time. The final break down takes $O(k \cdot n \cdot \binom{n}{k} 2^{n-k})$ time. Thus, the overall time is

$$\begin{aligned} T &= O(3^n) + O(n \textstyle\sum_{i=k+1}^{d} \binom{n}{i} 2^{n-i}) + \\ & \quad O(k \cdot n \cdot \binom{n}{k} 2^{n-k}) \\ &= O(3^n) + O(n \cdot 3^n) + O(n \cdot k \binom{n}{k} 2^{n-k}) \\ &= O(n \cdot 3^n + n \cdot k \cdot \binom{n}{k} 2^{n-k}) \end{aligned}$$

This is an improvement over the $O(2^{k+n-d_{max}} \binom{n}{k} \binom{n}{d_{max}})$ time algorithm given in [4]. Because this algorithm selects the exact same processors as the original bipartite algorithm, it's allocation performance (miss ratio, etc.) should be the same.

### 3.2 Discussion on Parallelization

This process can be easily parallelized on a CREW PRAM machine, by having each processor handle one x value. The processors each take an x value. They can then determine the y value associated with them, and wait until their y value has been processed, and then do their processing. If a total of $2^n$ processors are used, then the overall running time is $O(n \cdot 2^n)$. Using $3^n$ processors, each processor can wait until the two subcubes they represent have been calculated and then set their own value appropriately. This can be done in $O(n^2)$ time ($O(n)$ time for a processor to do its calculations, and it takes n of these to propagate the values all the way through the subcubes). Note that neither of these parallelizations are efficient.

On a more modest model, such as a hypercube, parallelization is more difficult. Here, each processor retains knowledge only of the availability of subcubes where the number of the processor is the lowest in the subcube (i.e., all the "don't cares" are 0). Start with the processors knowing whether they are on or not. Then, transmit to all it's neighbors whether it is on or not. The processors that receive that a neighbor is on, and they are lower numbered than that neighbor create the subcube containing them and their neighbor. Then, the processors transmit all the 1-d subcubes it knows about to it's neighbors. If a processor receives a 1-d subcube that it has a 1-d subcube that differs from that cube in exactly 1 position, and the processor would be the smallest numbered processor in that subcube, then it retains the 2-d subcube made

*Algorithm 3 :* Bipartite adaptation
/* Q = $3^n$ boolean array showing available subcubes
S = auxiliary array of start of blocks of subcubes
R = $3^n$ array of integers, initialized to 0 */
begin
$d_{max}$ = maximum dimension subcube available
count = $2 \cdot 3^{n-1}$
position = $2 \cdot 3^{n-1}$
/* fill down to dimension k+1 */
$R_i = 1$ for available subcubes of dimension $d_{max}$
for d = $d_{max}$ down to k+2, the requested dimension
for x = each number $< 2^n$ with d 1's
pos = $S_x$
b = highest position with a 1 in x
y = x with b set to 0
for i = 0 to $2^{n-d+1} - 1$
if the $(b-d+1)$-th bit of i is FALSE
if $(Q_{pos})$
$R_{i+S_y} = R_{i+S_y} + R_{pos}$
$R_{i+2^{(b-d+1)}+S_y} = R_{i+2^{(b-d+1)}+S_y} + R_{pos}$
pos = pos + 1
endif
endfor
endfor
endfor
d = k+1
/* the last split has be done along every dimension,
instead of just the highest "don't care" bit position */
for x = each number $< 2^n$ with $(k+1)$ 1's
/* bcnt - count of 1 bits of x right of bit b */
bcnt = 0
pos = $S_x$
for each position b in x that is 1, from highest
y = x with b set to 0
for i = 0 to $2^{n-k} - 1$
if the $(b-d+1+\text{bcnt})$-th bit of i is FALSE
$R_{i+S_y} = R_{i+S_y} + R_{pos}$
$R_{i+2^{(b-d+1+bcnt)}+S_y} = R_{i+2^{(b-d+1+bcnt)}+S_y}$
$+R_{pos}$
pos = pos + 1
endif
endfor
bcnt = bcnt + 1
endfor
endfor
Find subcube of requested dimension with smallest $R_i$
end

by joining the two 1-d cubes. This process is continued trading 2-d subcubes, 3-d subcubes, etc. until n-d subcubes are done. For each subcube that a subcube gets, it does a $O(n)$ operation on them, and perhaps a $O(n)$ transmit of the subcube (n dimensions implies n neighbors). A subcube can only lie in $2^n$ subcubes (either the subcube has a "don't care" in a position, or the bit matches the processor it's in). Thus, the run time of the parallel algorithm is $O(n \cdot 2^n)$ and utilizes $O(2^n)$ processors. This is exactly the same complexity as obtained on the CREW PRAM model using that many processors.

Note that these parallel versions are equivalent to, if not inferior to, previous parallel schemes that completely recognize available subcubes, such as multiple-GC [1].

## 4 Conclusions

This paper proposes a new algorithm for subcube recognition that runs in $O(n \cdot 3^n) = O(\log_2 P \cdot P^{\log_2 3})$ time, where n is the dimension of the overall hypercube and P is the number of processors. This is an improvement over previous algorithms which solves the same problem in about $O(4^n)$ time, or $P^2$. In addition, allocation schemes can be derived from this algorithm that have a better complexity than previous, equivalent algorithms, such as the bipartite algorithm [4]. Parallel versions of the proposed algorithm was also presented for the CREW PRAM model and the hypercube which ran in $O(n \cdot 2^n)$ on $2^n$ processors, and in $O(n^2)$ time on $3^n$ processors for the CREW PRAM model.

The algorithm has been implemented and shown to work. However, an empirical comparison of run-time and allocation efficiency remains an issue for future study for both the "first match" strategy and the bipartite adaptation of the subcube building algorithm.

## References

[1] M. Chen and K. G. Shin, "Processor Allocation in an N-Cube Multiprocessor Using Gray Codes," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1396-1407, Dec. 1987.

[2] S. Dutt and J. P. Hayes, "Subcube Allocation in Hypercube Computers," *IEEE Transactions on Computers*, vol. 40, no. 3, pp. 341-352, March 1991.

[3] P. Chuang and N. Tzeng, "A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers," *IEEE Transaction on Computers*, vol. 41, no. 4, pp 467-479, April 1992.

[4] B. Abali, F. Özgüner, and Cevdet Aykanat, "Dynamic Subcube Allocation in a Hypercube Multiprocessor," *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pp. 269-272, 1989.

[5] Y. Chang and L. N. Bhuyan, "Parallel algorithms for hypercube allocation," *Proceedings of Seventh International Parallel Processing Symposium*, p. 858, Newport, CA, April 1993.

[6] J. Kim, C.R. Das, and W. Lin, "A Top-Down Processor Allocation Scheme for Hypercube Computers," *IEEE Transactions on Parallel and Distributed Systems*, vol.2, no.1, pp. 20-30, Jan. 1991.