Partitioning on the Banyan-Hypercube Networks

Abdelghani Bellaachia Abdou Youssef

Department of Electrical Engineering and Computer Science
The George Washington University
Washington D.C. 20052

Abstract

Partitioning interconnection networks is one of the most important factors in increasing resource utilization of parallel machines. This paper addresses the partitioning of the Banyan-Hypercube networks (BH's). BH's have been recently introduced as new fixed interconnection networks. It was shown that they have many hypercube features such as self-routing and efficient embedding of rings and meshes. In addition, they have advantages over hypercubes in extendibility, diameter, average distance, and embedding of hierarchical structures such as trees, pyramids and multiple pyramids. BH's are also highly partitionable.

Partitioning strategies as well as data structures for partitioning are proposed and studied. Simulation results of internal, external, and total fragmentations for uniform and exponential distributions of request sizes are presented and discussed. The buddy system strategy of partitioning the hypercube is also simulated for comparison purposes. BH exhibits a better internal fragmentation over the hypercube, and for large request sizes the total fragmentation of the two networks are comparable. It is also shown that the internal fragmentation in BH decreases as the number of levels of BH increases.

1-Introduction

Highly parallel computers provide great computational power to solve problems previously considered impractical. However, the availability of several processing elements is not enough. The resources should be well utilized and managed. One way of improving performance is system partitioning [3][6][7][10]. Partitioning consists of splitting the network into several subnetworks, called partitions, of different sizes. Each partition must have the same structure as the original network in order to use the same routing and mapping algorithms. Partitionable parallel computers can provide an increase in system throughput as well as in the speedup of a single job [4]. The Throughput is increased by executing several jobs simultaneously. The speedup of a single job is maximized by executing its tasks simultaneously. Examples of partitionable systems include : the Ncube[6], PASM[8], Connection Machine[3], and Cosmic Cube[10].

The Banyan-Hypercube (BH) networks were first introduced as new fixed interconnection networks in [13]. It was shown that they have many hypercube features such as self-routing and efficient embedding of rings and meshes. In addition, they have advantages over hypercubes in extendibility, diameter, average distance, and embedding of hierarchical structures such as trees, pyramids and multiple pyramids. It was also shown that BH's have a great partitioning flexibility. Partitioning the BH is the focus of this paper.

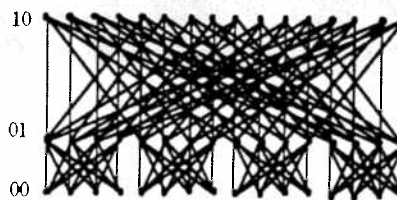
Network partitioning needs: (1) a data structure to keep track of available partitions in the system; (2) a partition allocation strategy that partitions the network and allocates partitions to requests; and (3) a deallocation strategy that merges freed partitions with other free partitions in the system to form larger partitions and minimize fragmentation.

The paper is organized as follows. The next section gives an overview of BH's, and other definitions related to partitioning. Section 3 discusses the data structures used to partition the BH and analyzes their space complexity. In section 4, algorithms for partitioning strategies of the banyan-hypercubes are presented. Section 5 presents metrics to measure internal, external, and total fragmentation in the system. Experimental results of all strategies are given in section 6. The last section gives concluding remarks and future directions.

2-Preliminaries and Definitions

An extended definition of the banyan-hypercubes as well as properties related to partitioning are presented in this section. Banyan-hypecubes are a synthesis of banyans and hypercubes. They were first defined in [13], as a new topology. Banyans are reviewed first.

A banyan is a Hass diagram of a partial ordering where there is a unique path from every base to every apex[4]. A base is any vertex having no arcs incident into it, and an apex is any node having no arcs incident out from it. An *L-level* banyan is a banyan whose nodes can be arranged into L levels so that the arcs are only between adjacent levels. A regular banyan is an *L-level* banyan where all the nodes except the bases have the same indegree F called the *fanout* and all the nodes except the apexes have the same outdegree S called the *spread*. A rectangular banyan is a regular banyan where $S=F$. In this case, all the levels have the same number of nodes, which is S^{L-1}. Figure 1 shows a rectangular banyan of spread 2 and another of spread 4.



(a). Banyan of spread 4

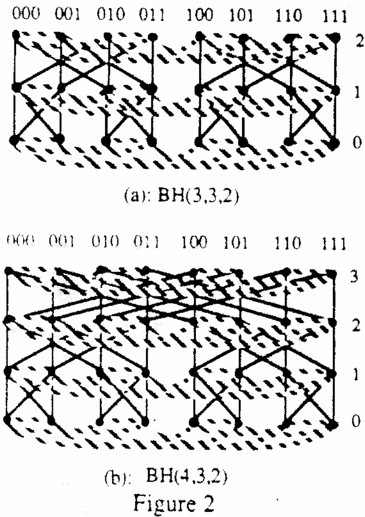
(b). Banyan of spread 2

Figure 1

CH2908-2/90/0000/0343/$01.00 – 1990 IEEE

A banyan-hypercube, denoted BH(h,k,s), where $h \leq k+1$ and s is power of 2, is the first h levels (from the base) of a $(k+1)$-level rectangular banyan of spread s, such that the nodes in each level are interconnected by a hypercube (dashed lines in figure 2). The levels of the banyan are numbered from 0 to h -1 and the nodes in each level are labeled in binary from 0 to s^k-1. Therefore, each node is labeled by a pair (L,X) where $0 \leq L \leq h$ -1 and X is a cube address of the form $x_{k-1}...x_1x_0$ in the number system of base s. The total number of nodes in the network is hs^k. For a complete definition of the banyan-hypercube, the reader is referred to [13]. If $h=k+1$, the network is called a full banyan-hypercube. Figure 2 shows banyan-hypercube networks.

It was proven in [14] that any l successive levels of a BH(h,k,s) where $1 \leq l \leq h$, constitute a graph which is isomorphic to BH(l,k,s). Therefore, any subnetwork of BH(l,k,s) of l_i levels, starting from level b_i, such that the nodes in each level are interconnected in a k_i-cube is a banyan-hypercube, called subbanyan-hypercube. Consequently, the parameters h, k, s are not enough to describe a subbanyan-hypercube and new parameters are needed to locate the subnetwork in the original network. One of these parameters is the starting level number, called the *base level*. For a BH(h,k,s), the base level is 0, therefore BH(h,k,s) can be referred to as BH$(0,h,k,s)$.



(a): BH(3,3,2)

(b): BH(4,3,2)

Figure 2

Definition 1: A subbanyan-hypercube BH(b_i,l_i,k_i,s) of BH$(0,h,k,s)$, where $0 \leq b_i \leq h$-1, $1 \leq l_i \leq h$, and $0 \leq k_i \leq k$, is a banyan-hypercube of l_i levels starting at level b_i such that the nodes in each level are interconnected in a k_i-cube. The base level b_i must satisfy the following conditions:

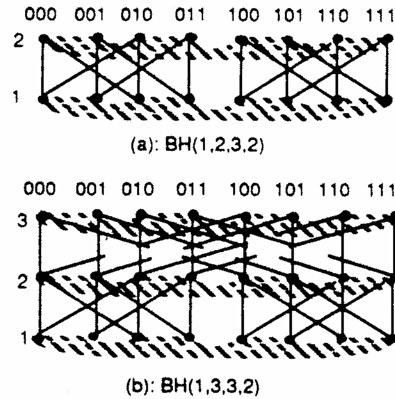
(1) $0 \leq b_i \leq k_i+1-l_i$ if $1 < l_i \leq h$ $(l_i \neq 1)$

(2) $0 \leq b_i \leq h$ -1 if $l_i=1$

Note that the labels of the levels of BH(b_i,l_i,k_i,s) are b_i, b_i+1, b_i+2,..., b_i+l_i-1. Throughout this paper b_i is the *base* of the subbanyan-hypercube, k_i is the *logarithmic width*, s^{k_i} is the *width*, and h is the *height*. Figure 3 shows two subbanyan-hypercubes.

Definition 2: A (l_i,k_i)-partition is any subbanyan-hypercube whose height is l_i and logarithmic width k_i, where $l_i \leq k_i+1$. The size of an (l_i,k_i)-partition is $l_i\, s^{k_i}$.

For practical considerations, the value of s is preferred to be 2 or 4. Throughout this paper only banyan-hypercubes with spread s=2 are treated. In this case each node in the network is identified by its level and its cube-address $x_{k-1}...x_1x_0$, where x_i is either 0 or 1.



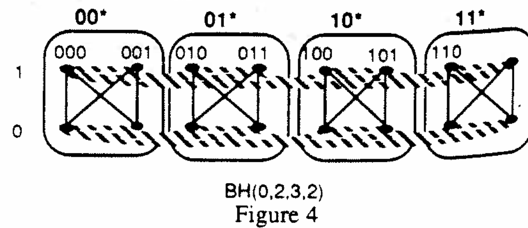
(a): BH(1,2,3,2)

(b): BH(1,3,3,2)

Figure 3

One can easily see that a partition BH$(b_i,l_i,k_i,2)$ is still not uniquely identified. This is because for a given base b_i, and a number of levels l_i, we might have 2^{k-k_i} k_i-cube in a k-cube. In figure 4 the BH(0,2,3,2) has 4 identical subbanyan-hypercubes, BH(0,2,1,2). To uniquely identify partitions BH$(b_i,l_i,k_i,2)$ of a BH$(0,h,k,2)$, a representation based on the ternary representation in [3] is used to address the k_i-cube of each partition. This will be referred to as the cube address of the partition. The address of each k_i-cube is represented by k digits, where k is the logarithmic width of BH, such that each digit is 0, 1, or *, where * is a "don't care". For example, the subcube of the nodes {000,100,001,101} in the 3-cube has the address *0* (2-cube). When using the buddy system strategy in the allocation of subcubes in a hypercube [3], all the *'s constitute the lower part of the address and are referred to as $*^{k_i}$, where k_i is the size of the subcube. We refer to this addressing scheme as the *buddy addressing mode*.

The BH(0,2,3,2) of figure 4 has the cube address $*^3$ (i.e ***) and its subbanyan-hypercubes BH(0,2,1,2) have the following cube addresses, from left to right, 00*, 01*, 10*, 11*.

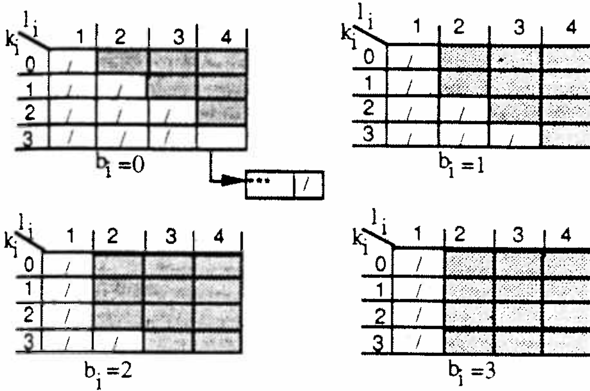


BH(0,2,3,2)

Figure 4

3-Data Structures for Network Partitioning

During partitioning, some partitions may become free either when a given task completes its execution or when splitting a partition into smaller partitions to accommodate an incoming task with a small request. In either case, a data structure is needed to record available partitions. Two data structures are considered for the partitioning of the banyan-hypercube.



Data structure for BH(0,4,3,2)
Figure 5

Since each partition in the network is characterized by its base level, its number of levels, its logarithmic width, and its cube address, a three-dimensional array, **P**, is first considered, where each entry points to a linked list of free partitions. The nodes of the linked list pointed to by the entry (b_i,k_i,l_i) contain the cube addresses of those partitions that have the base level b_i, the number of levels l_i, and the logarithmic width k_i. Initially, all entries point to null, except for the entry **P**$[0,k,l]$, corresponding to the original network. Figure 5 shows the data structure of the BH(0,4,3,2).

The not-used entries (shaded area) in the array correspond to the subnetworks which do not satisfy either conditions (1) or (2) defined in the previous section. As can be seen in the example given in figure 5, the array is very sparse, less than 50% of the entries is used. The space complexity of **P** is clearly O($h^2.k$.O(**list_space**)), where O(**list_space**) is the space used by the largest list in the system. Before the evaluation of O(list_space), we first give the second data structure, since its space complexity includes the same term. This second data structure is a mapping of the 3-dimensional array **P** into a one-dimensional array, called **PART**. The size of **PART** is equal to the number of the used entries in **P**. The following proposition gives the total number of used entries in the case of a full banyan-hypercube.

Proposition 1: The total number, U, of used entries used in P for a full banyan-hypercube BH(0,h,k,2), where $h=k+1$, is:

$$U=(k+1)^2+\frac{1}{6}k(k+1)(k+2)$$

Note that for a full BH(0,11,10,2) only 28.2% of the space of the first data structure is used.

The mapping function of the element of **P** into the element of **PART** is derived by a rowwise counting of the occupiable (i.e. non-shaded) entries of the 3-dimensional array **P**. It is given by the following proposition:

Proposition 2: The entry **P**$[b_i,k_i,l_i]$ of a given partition BH(b_i,l_i,k_i,2) in a BH(0,h,k,2) is mapped into **PART** using the following function, **F**:

$$F[b_i,k_i,l_i]=\begin{cases} f(b_i,k_i) & \text{if } l_i=1 \\ f(b_i,k_i)+l_i+b_i+(k-b_i)l_i-\frac{1}{2}l_i(l_i-1)-(k+1) & \text{if } l_i\neq 1 \end{cases}$$

Where

$$f(b_i,k_i)=k_i+(k+1)b_i+\frac{1}{2}(h(h+1)b_i-\frac{1}{2}(2h+1)b_i(b_i+1)+\frac{1}{6}b_i(b_i+1)(2b_i+1))$$

The address of an available partition BH(b_i,l_i,k_i,2) is then located in the list pointed to by **PART**[**F**$[b_i,k_i,l_i]$]. The space complexity of **PART** is now reduced to O(U.O(list_space)), where U is given by proposition 1. To compute the term O(list_space) which is the space of the largest list in the system, theorem 1 is needed. The following lemmas are useful in the proof of theorem 1.

Lemma 1: The number of k_i-cubes, using the buddy system, in a hypercube of dimension k is 2^{k-k_i}, and the total number of subcubes is $2^{k+1}-1$.

Proof: The proof is straightforward and hence omitted. The interested reader can find the proof in [3]

■

Lemma 2: Given a BH(0,h,k,2), the total number $K_h(l_i,k_i)$ of (l_i,k_i)-partitions having their top level coincide with the top level of the BH is:

$$K_h(l_i,k_i)=\begin{cases} 2^{k-k_i} & \text{if } l_i=1 \\ 0 & \text{if } l_i\neq 1 \text{ and } k_i<h-1 \\ 2^{k-k_i} & \text{if } l_i\neq 1 \text{ and } k_i\geq h-1 \end{cases}$$

Proof: Case 1: $l_i=1$, that is, each (l_i,k_i)-partition is a hypercube of dimension k_i. The total number of subcubes of dimension k_i at level h-1 is 2^{k-k_i} by lemma 1.

Case 2: $l_i\neq 1$. In this case, it can be seen that the number of (l_i,k_i)-partitions is independent from the value of b_i. Therefore, it is sufficient to count the number of partitions where $b_i=0$. Let LBH(0,l_i,k_i,2) be a partition of BH(0,h,k,2). The top level of LBH coincides with the top level of BH only if $b_i+l_i-1=h-1\leq k_i$. Thus, if $k_i<h-1$, no such partitions exist. On the other hand, if $k_i\geq h-1$ then BH can be built from 2^{k-k_i} copies of LBH. Therefore, $K_h(l_i,k_i)=2^{k-k_i}$.

■

Theorem 1: The total number of (l_i,k_i)-partitions in a BH(0,h,k,2) is:

$$P_{h,k}(l_i,k_i)=\begin{cases} (k_i-l_i+2)2^{k-k_i} & \text{if } l_i\neq 1 \\ h2^{k-k_i} & \text{if } l_i=1 \end{cases}$$

Proof: Let $P_{h,k}(l_i,k_i)$ and $P_{h-1,k}(l_i,k_i)$ be the total number of (l_i,k_i)-partitions in BH(0,h,k,2) and in BH(0,h-1,k,2), respectively. $P_{h,k}(l_i,k_i)$ can be computed using the following recurrence relation:

$$P_{h,k}(l_i,k_i)=P_{h-1,k}(l_i,k_i)+K_h(l_i,k_i)$$

Where $K_h(l_i,k_i)$ is the number of (l_i,k_i)-partitions such that their top level coincide with the top level of BH(0,h,k,2) as in lemma 2. Letting $a_m=P_{m,k}(l_i,k_i)$, we have: $a_m=a_{m-1}+K_m(l_i,k_i)$, where m is the number of levels in the network. Two cases are considered:

Case 1: $l_i=1$. In this case the value of $K_m(l_i,k_i)$ is 2^{k-k_i} and the recurrence relation becomes:

$$a_m=a_{m-1}+2^{k-k_i}$$

The initial value of this relation is when $m=1$, that is the hypercube case. This value is computed by lemma 1 and is $a_1=2^{k-k_i}$. The solution of this relation is simple and the result is: $a_m=m2^{k-k_i}$, and therefore $P_{h,k}(l_i,k_i)=a_h=h2^{k-k_i}$

Case 2: $l_i\neq 1$. Lemma 1 introduces two sub-cases to compute the value of $K_m(l_i,k_i)$ in BH(0,h,k,2) when $l_i\neq 1$:

$$K_m(l_i,k_i)=\begin{cases} 0 & k_i<m-1 \\ 2^{k-k_i} & k_i\geq m-1 \end{cases}$$

Therefore, two different recurrence relation result from the above conditions:

$$a_m = a_{m-1} \quad \text{if } m > k_i + 1 \quad (1)$$
$$a_m = a_{m-1} + 2^{k-k_i} \quad \text{if } m \leq k_i + 1 \quad (2)$$

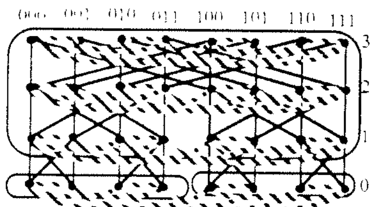
The initial value for these two relations can be derived by observing that the height of the smallest BH network which has (l_i,k_i)-partitions must be $\geq l_i$. Therefore, m can take values between l_i and h, where h is the height of the network. The value of a_{l_i} can be computed by applying lemma 2 to BH$(0,l_i,k,2)$, leading to $a_{l_i}=2^{k-k_i}$, since $l_i-1 \leq k_i \leq k$. From relation (2) we conclude that $a_m=a_{l_i}+(m-l_i)2^{k-k_i}$ for $m \leq k_i+1$. In particular, $a_{k_i+1}=a_{l_i}+(k_i+1-l_i)2^{k-k_i}$. From relation (1) we have $a_h=a_{h-1}=...= a_{k_i+2}=a_{k_i+1}$. Therefore, $a_h = a_{k_i+1} = a_{l_i}+(k_i+1-l_i)2^{k-k_i} = 2^{k-k_i}+(k_i+1-l_i)2^{k-k_i} = (k_i-l_i+2)2^{k-k_i}$. ■

Now, we are in a position to compute the term O(list_space). It is clearly seen that the lower bound is attained when the whole network consists of one free partition, (figure 5). From Theorem 1 we can see that for small values of l_i and k_i, the number of partition $P_{h,k}(l_i,k_i)$ is large. So, for $l_i=1$, $k_i=0$, and all the base levels b_i, the total number number of (1,0)-partition is computed by theorem 1, $P_{h,k}(1,0)= h2^k$. In the next section we will see that when using the buddy system we may end up having only half of the processors allocated in a given level. Therefore, for a given b_i, we may have 2^{k-1} free processors in the worst case. In that case the space needed by the list is O(list_space)=O(2^{k-1}). Hence, the space complexity of **PART** is O($U.2^{k-1}$) and that of **P** is O(h^2k2^{k-1})=O(hkN), where U is given in proposition 1 and $N=h2^k$.

4-Partitioning Strategies

Partitioning a parallel architecture into several partitions of different sizes to accommodate incoming tasks is a problem similar to memory management in a uniprocessor. It allows multitasking where multiple tasks can be executed simultaneously in the system. Efficient partitioning strategies are an important factor in higher performance and better utilization of systems.

To partition the banyan-hypercube, a dynamic two-way partitioning system (**TWPS**) is proposed. In [13], the banyan-hypercube was shown to be partitioned along two directions: horizontally and vertically. The vertical partitioning uses the binary buddy system [13], and it is possible only if the partition, BH$_i(b_i,l_i,k_i,2)$, is not a full banyan-hypercube: $b_i < k_i + 1 - l_i$ (condition (1) of definition 1). The horizontal partitioning was limited to the top of a full banyan-hypercube. This reduces the flexibility of partitioning the network. Our **TWPS** schema uses the same vertical partitioning, but it does not restrict the horizontal partitioning to the top level of a full banyan-hypercube. Any partition can now be split at any level. This makes our partitioning more flexible: more partitions can be allocated. Figure 6 shows an example of horizontal and vertical partitioning.



BH(0,4,3,2)

Figure 6: horizontal and vertical partitioning

Some partitions can be split or combined horizontally and vertically, others can only be split (or combined) horizontally. For example if the network is full, it cannot be split vertically.

During the process of splitting and combining partitions, we are interested in valid partitions. Formally, a partition BH$_i(b_i,l_i,k_i,2)$ in a BH$(0,h,k,2)$ is a valid partition if its cube address is of the form $a_{k-1}a_{k-2}...a_{k-k_i+1}*^{k_i}$ and if b_i, l_i, and k_i satisfy conditions (1) and (2) in definition 1.

A valid partition BH$_i(b_i,l_i,k_i,2)$ with cube address $a_{k-1}a_{k-2}...a_{ki}*^{k_i}$ can be split vertically into two valid partitions of the form BH$_i(b_i,l_i,k_i-1,2)$ with cube addresses $a_{k-1}a_{k-2}...a_{ki}0*^{k_i-1}$ and $a_{k-1}a_{k-2}...a_{ki}1*^{k_i-1}$. The two partitions are called cube-buddies.

For every l in the range between 1 and l_i-1, BH$_i(b_i,l_i,k_i,2)$ can be split horizontally into two partitions BH$_l(b_i,l,k_i,2)$ and BH$_u(b_i+l,l_i-l,k_i,2)$, both of cube address $a_{k-1}a_{k-2}...a_{ki}*^{k_i}$. BH$_l$ and BH$_u$ are called level-buddies; BH$_l$ is the lower level-buddy and BH$_u$ is the upper level-buddy.

Combining, the opposite process of splitting, can be done horizontally and vertically as follows. If BH$_i(b_i,l_i,k_i,2)$ and BH$_j(b_j,l_j,k_j,2)$ are valid partitions with cube addresses Ai=$a_{k-1}a_{k-2}...a_{ki}*^{k_i}$ and Aj=$b_{k-1}b_{k-2}...b_{kj}*^{k_j}$, then:

- if $b_i=b_j$ and $l_i=l_j$ and $k_i=k_j$ and Ai and Aj agree in all but bit position k_i then the two partitions (cube-buddies) can be combined into a single partition BH$(b_i,l_i,k_i+1,2)$ with cube address $a_{k-1}a_{k-2}...a_{ki+1}*^{k_i+1}$.
- if $k_i=k_j$ and $b_j=b_i+l_i$ and the two cube addresses are identical, then the two partitions are level-buddies and can be combined into a single partition BH$(b_i,l_i+l_j,k_i,2)$ with cube address $a_{k-1}a_{k-2}...a_{ki}*^{k_i}$

Our **TWPS** system of the banyan-hypercube consists of three steps:

step1 - Computation of the **size of the partition** required for the incoming task.
step2 - **Allocation** of a partition to the incoming task.
step3 - **Recombining** partitions which have been released when tasks complete their execution, or partitions resulting from splitting larger partitions during the allocation step.

4.1 Computation of the partition size:

For an incoming task of size n, the computation of the size of the required partition is such that the internal fragmentation is minimized. The logarithmic width k_t and the height l_t of the required partition are such that $n \leq l_t 2^{k_t}$. For a given value of n there may be more than one solution for l_t and k_t. For example, for a task of size n=57 the smallest partition to accommodate this task has 64 processors which may be expressed in three different ways: l_t=1 and k_t=6, or l_t=2 and k_t=5, or l_t=4 and k_t=4. A preferable choice of l_t and k_t is such that l_t is the smallest value among all possible values. In the case of n=57, that choice is l_t=1 and k_t=6. This is because the total number of (l_i,k_i)-partitions is greater than the number of (l_j,k_j)-partitions of same size, if $l_i<l_j$ and $k_i>k_j$, as shown in theorem 2.

Theorem 2: The total number of (l_i,k_i)-partition is greater than the total number of (l_j,k_j)-partition of the same size (i.e $l_i2^{k_i}=l_j2^{k_j}$), if $l_i<l_j$ and $k_i>k_j$

Proof: Let p_i and p_j be the number of (l_i,k_i)-partitions and (l_j,k_j)-partitions, respectively. We will prove that $p_i>p_j$. The proof is by contradiction.

Case 1: l_i=1. By theorem 1, the values of p_i and p_j are $h2^{k-k_i}$ and $(k_j-l_j+2)2^{k-k_j}$, respectively, since $l_i=1$ and $l_j>l_i$ ($l_j\neq1$). Let us assume that $p_i \leq p_j$, then we have $h2^{k-k_i} \leq (k_j-l_j+2)2^{k-k_j}$. Since $k_j<k_i$, it follows that $2^{k-k_i}<2^{k-k_j}$, and therefore $h<k_j-l_j+2$ (or $h-1<k_j-l_j+1$). Recall that h-1 is the top level of the BH and k_j-l_j+1 is the highest possible base of (l_j,k_j)-partitions. So, h-1 should be greater than or equal to k_j-l_j+1, contradicting the fact that $h-1<k_j-l_j+1$.

Case 2: $l_i\neq1$. The proof in this case is similar to case 1. ■

To compute l_t and k_t for a given n, the procedure FIND and FIND_SMALLEST are given below. First the procedure FIND computes one possible value of l_t and k_t, then the procedure FIND-SMALLEST finds the value with the smallest height.

```
Procedure FIND(n, l_t, k_t)
    m:integer;
    m=1;
    while 2^m ≤ n do
      l_t = n/2^m
      if l_t ≤ m+1
      then
          k_t = m;
          return;
      else
          m = m+1;
      endif
    endwhile
end FIND;

Procedure FIND-SMALLEST(l_t, k_t)
    p,q:integer;
    p=1; q=k_t+1;
    while (p<l_t) do
      if p2^q = l_t 2^{k_t}
      then
          l_t=p; k_t=q;
          return;
      else
          p=p+1;
          q=q+1;
      endif
    endwhile
end FIND-SMALLEST;
```

The procedure FIND takes O(log$n$), while FIND-SMALLEST takes O(h), where n is the size of the incoming task and h is the height of the original network. Therefore, the total time is O(h+log$n$).

4.2 Allocation algorithms

The **TWPS** always starts allocating partitions from the top available partitions in the network (i.e. high base levels). This is because all the top partitions with a higher base can be allocated at smaller base level but not vice-versa. For example in figure 2 (b) we can always allocate a (2,3)-partition either at base 0 or at base 2, but a (2,1)-partition can only be allocated at base 0. Once the value of l_t and k_t are computed, the possible values of the base b_t of the (l_t,k_t)-partitions are in the range from 0 to k_t-l_t+1 if $l_t \neq 1$, and in the range from 0 to h-1 if l_t=1 (definition 1). For each value of b_t, starting from the largest value, the **TWPS** allocates the first free partition in the list pointed to by the entry **PART**[**F**(b_t,l_t,k_t)]. If there is no available partition for any possible value of b_t, then a larger partition is to be searched. There are two possible ways to choose a larger partition: (1) A partition with a larger logarithmic width which when vertically split, yields cube-buddies; one of these cube-buddies will be granted to the request, the other will be returned to the data structure; (2) A partition with a larger number of levels which, when horizontally split, yields level-buddies; one of these level-buddies will be granted to the request, the other will be returned to the data structure. Therefore, two allocation strategies are considered: Cube-Buddy First Allocation (CBFA) and Level-Buddy First Allocation (LBFA).

Cube-Buddy First allocation (CBFA) :

This allocation procedure starts looking for a free (l_i,k_i)-partition, for all possible values of b_i, where initially $l_i = l_t$ and $k_i = k_t$. If there is a free partition, then it is allocated; otherwise, for each possible value of l_i between l_t and h, the logarithmic width k_i is incremented by 1 (up to k) searching for (l_i,k_i)-partition. The search stops when a partition is found. If there is no free partition, then the request is queued for later scheduling, a topic outside the scope of this paper. The algorithm of this strategy is given below.

```
Procedure CBFA(b_i,l_i,k_i,l_t,k_t)
/* l_t and k_t are the number of levels and the logarithmic width */
/* of the partition of the incoming task. */
/* found: flag set to true if there is an available partition */
/* If a free partition BH_i is found, then it is returned */
          l_i=l_t ; k_i=k_t
          found=false;
step1:    compute the set B of the possible base levels b_i of
          (l_i,k_i)-partitions:0≤b_i≤k_t-l_t+1 if l_t≠1 and 0<b_i≤h-1
          if l_t=1
step 2:   while (l_i<=h) do
step 2.1:   for each b_i in B do
                if there is an available partition,
                BH(b_i,l_i,k_i,2), at the base level b then
                    found=true and exit. /*exit while loop */
                endif
            endfor
step 2.2:   /* Look for a larger partition */
            for each base level b_i in B do
                check all the (l_i,k_j)-partitions, where k_i<k_j≤k
                If there is an available partition then
                    found=true;k_i=k_j ;exit; /* exit while loop */
                endif
            endfor
            l_i=l_i+1;
          endwhile
step 3:   if found then
              SPLIT (b_i,l_i,k_i);
          else
              put request in the waiting queue and found=false
          endif
end of CBFA;
```

The performance of CBFA is dominated by step2. Step 2.1 takes O(h) since the value of b_i range from 0 to h-1 (case l_i=1). Step 2.2 takes O($h.k$) since finding an (l_i,k_i)-partition takes O(k). Therefore, the overall time of CBFA is $O(h.h.k)=O(h^2k)$. If we have a full Banyan-Hypercube, $h=k+1$, then the number of steps becomes $O(k^3)$. If N is the number of processors, $N=(k+1)2^k$, then k is equal to $log(N/(k+1))$, henceforth, we have $O(log^3(N/k))$.

Level-Buddy First allocation (LBFA):

This allocation is similar to cube-buddy first allocation, except that partitions with more levels are first searched before a partition with a larger logarithmic width. As with CBFA, (l_i,k_i)-partitions are searched, for all possible values of b_t, where initially $l_i=l_t$ and $k_i=k_t$. If there is a free (l_i,k_i)-partition, then it is allocated; otherwise, for each possible values of k_i between k_t and k, the number of levels is incremented by 1 (up to h) searching for an (l_i,k_i)-partition. The search stops when a partition is found. If there is no free partition the request is queued.

The performance of LBFA is similar to the performance of the CBFA procedure. Step2.1 is the same in both procedures. Step 2.1 in LBFA takes $O(h^2)$ since we looking for an

(l_i,k_i)-partition with larger height. The total time is then $O(k.h^2)$. Similarly to the performance analysis of CBFA, for a full banyan-hypercube the time becomes $O(log^3(N/k))$.

```
Procedure LBFA(b_i,l_i,k_i,l_t,k_t)
/* l_t and k_t are the number of levels and the logarithmic width */
/* of the partition of the incoming task. */
/* found: flag set to true if there is an available partition */
/* If a free partition BH_i is found, then it is returned */
          l_i=l_t ; k_i=k_t
          found=false;
step1:    compute the set B of the possible base levels b_i of
          (l_i,k_i)-partitions:0≤b_i≤k_t-l_t+1 if l_t≠1 and 0<b_i≤h-1
          if l_t=1
step 2:   while (k_i≤k) do
step 2.1:    for each b_i in B do
                if there is an available partition,
                BH(b_i,l_i,k_i,2), at the base level b_i then
                   found=true and exit. /* exit while loop */
                endif
             endfor
step 2.2:    for each base level b_i in B do
                check all the (l_j,k_i)-partitions, where
                l_i<l_j≤k_i+1-b_i if the network is full; otherwise
                l_i<l_j≤h.
                If there is an available partition then
                   found=true;l_i=l_j; exit; /* exit while loop */
                endif
             endfor
             k_i=k_i+1;
          endwhile
step 3:   if found then
             SPLIT (b_i,l_i,k_i)
          else
             put task in the waiting queue and set found=false
          endif
end of LBFA;
```

4.3 Splitting algorithm

The splitting of a large partition is the same for both allocation policies, CBFA and LBFA. Once a free partition BH_i (with cube address $a_{k-1}...a_{k_i}*^{k_i}$,say) is found either by CBFA or by LBFA the splitting algorithm compares the size of BH_i with the size of the required (l_t,k_t)-partition. If they are equal, it allocates the partition and stops; otherwise, BH_i is large and should be split. The splitting is first done horizontally yielding a number of partitions, one of which is of the desired height l_t, and denoted BH_r. The other resulting partitions are returned to the data structure, using the RECOMBINE procedure in the algorithm below. If the logarithmic width of BH_r is equal to k_t then BH_r is allocated; otherwise it is split vertically k_i-k_j times, where $1\le k_j\le k_i$. All the resulting partitions will be returned to the data structure except for one partition, the one with cube address $A_i=a_{k-1}...a_{k_i}z_{k_i-1}...z_{k_t}*^{k_t}$ where all the z's are 0's. That partition is granted to the request. The splitting algorithm is given below. The RECOMBINE procedure refers to the recombining procedures given in the next subsection.

The complexity of the splitting algorithm is determined by the time of step 2 and the time of step 3. Step 2 consists of first locating an available level_buddy partition which takes $O(h)$ in the worst case (case $l_i=1$), and second calling RECOMBINE procedure which will be shown to take $O(k)$. Thus, step 2 takes $O(h+k)=O(k)$, since $h\le k+1$. Step 3 takes $O(k)$, the time needed by the buddy system to split a given partition. Therefore, the overall time of SPLIT is $O(k)$.

```
Procedure SPLIT(b_i,l_i,k_i )
/* If BH_i is large and l_i > l_t then let b_t be the base of the */
/* required partition. The value of b_t is such that b_t≥b_i and */
/*b_t+l_t-1≤top_i , where top_i=b_i+l_i-1 is the top level of the */
/* allocated partition. Note that b_t +l_t-1=top_t is the top level */
/* of the required partition. */
        top_i , top_t ,b_t: integer;
        top_i=b_i+l_i-1;top_t =b_t +l_t-1;
step 1: if l_i=l_t and k_i=k_t then
           allocate the partition BH(b_i,l_t,k_t,2) and stop
        endif
step 2: if l_i≠l_t then
           Find the largest value of b_t such that b_i≤b_t ≤k_t -l_t +1
           and top_t=b_t+l_t -1≤top_i.
           if top_i >top_t then /* recombine upper level-buddy */
              RECOMBINE(top_t+1,top_i-top_t,k_i)
           endif
           if b_t>b_i then      /* recombine lower level-buddy */
              RECOMBINE(b_i,b_t-b_i,k_i)
              b_i=b_t
           endif
        endif
step 3: if k_i≠k_t then
           split vertically, using the buddy system, until k_i=k_t.
        endif
step 4: allocate BH(b_i,l_t,k_t) to the incoming task..
end SPLIT;
```

4.4 Recombining algorithm

The third step of our partitioning system consists of recombining released partitions with available ones in order to reduce the total fragmentation in the system. Since each partition may have a cube-buddy freed partition or a level-buddy freed partition, two recombining policies are considered.

Cube-buddy first recombining (CBFR):

Recombining is repeatedly done, first on all the free cube-buddy partitions then on the upper and lower level-buddy partitions. Figure 7 (a) shows when the cube-buddy first recombining is better. The recombining of released BH(0,2,2,2) whose cube address is 0** with the free BH(0,2,2,2) whose cube address is 1**, gives a partition, BH(0,2,3,2) whose cube address is ***, which has 16 processors, while if we recombined it with its upper level-buddy, we will end up with a partition with only 12 processors.

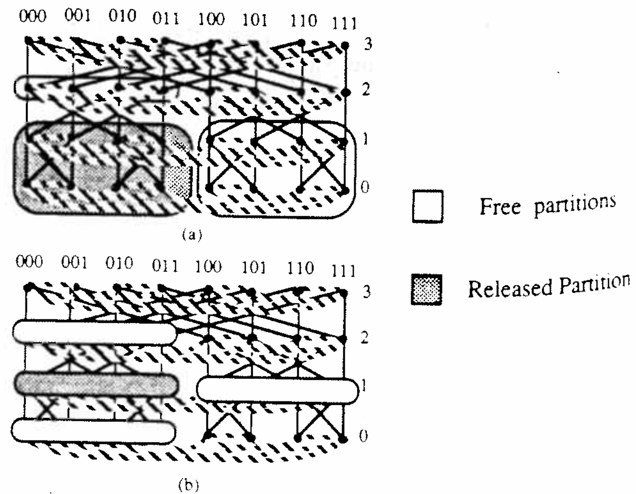


Figure 7

```
Procedure CBFR(b_r,l_r,k_r):
/* Let BH_r be the partition to be recombined */
        l_u,l_w,b_w,l_j,b_j: integer=0;
        cube_buddy_found, level_buddy_found:
        boolean=false;
start:  l_u=0,l_w=0,b_w=0;
step1   If there is a cube address of the cube-buddy partition
        of BH_r in the list pointed by PART[F(b_r,k_r,l_r)] then
            k_r=k_r+1; cube_buddy_found=true;
        else
step2       if the upper level-buddy of BH_r is in the list
            pointed to by PART[F(b_j,k_r,l_j)], whereb_j=b_r+l_r,
            and 1≤l_j≤k_r+1-b_j ( 1≤l_j≤h if BH_r is not full) then
                l_u=l_j ; level_buddy_found=true;
            endif
            if the lower level-buddy of BH_r is in the list
            pointed by PART[F(b_j,k_r,l_j)], where 0≤b_j≤b_r-1and
            l_j=b_r-b_j then
                b_w=b_j; l_w=l_j
            endif
            l_r=l_w+l_r+l_u;
            b_r=min(b_w,b_i)
        endif
step3   /* no more free partition to recombine */
        if cube_buddy_found=false and
        level_buddy_found=false then
            insert the cube address of BH(b_r,l_r,k_r,2) into the
            list of PART[F(b_r,k_r,l_r)]
        else
            goto start
end CBFR;
```

The time required to recombine a partition essentially depends on the state of the system: the available free partitions. In the worst case, recombining with free cube-buddy partitions, step 1, takes $O(k)$ and recombining with free level_buddy, step 2, takes $O(h)$. Therefore, the overall performance of CBFR procedure is $O(k+h)=O(k)$, since $h \le k+1$.

Level-buddy first recombining (LBFR):

Recombining is repeatedly done, first on all level-buddy partitions then with the cube-buddy partition. Figure 7 (b) shows when the level-buddy recombining is better. The recombining of BH(1,1,2,2) whose cube address is 0**, is first done with all the level-buddy partitions, which result in a partition of 12 processors. If the recombining is first performed with the cube-buddy partition, the resulting partition will only have 8 processors. The procedure for LBFR is similar to CBFR except that we first start recombining with the free level_buddy partition (i.e. interchange step 1 and step 2 in CBFR).

```
Procedure LBFR(b_r,l_r,k_r):
/* Let BH_r be the partition to be recombined */
        l_u,l_w,b_w,l_j,b_j: integer;
        cube_buddy_found, level_buddy_found: boolean=false;
start:  l_u=0,l_w=0,b_w=0;
step1   if the upper level-buddy partition of BH_r is in the list
        pointed by PART[F(b_j,k_r,l_j)], where b_j=b_r+l_r, and
        1≤l_j≤k_r+1-b_j ( 1≤l_j≤h if BH_r is not full) then
            l_u=l_j ; level_buddy_found=true;
        endif
        if the lower level-buddy partition of BH_r is in the list
        pointed by PART[F(b_j,k_r,l_j)], where 0≤b_j≤b_r-1and
        l_j=b_r-b_j then
            b_w=b_j; l_w=l_j ;level_buddy_found=true;
        endif
        l_r=l_r+l_w+l_u;
        b_r=min(b_w,b_i);
step2   If level_buddy_found=false then
            If the cube-buddy partition of BH_r is in the list
            pointed by PART[F(b_r,k_r,l_r)] then
                k_r=k_r+1; cube_buddy_found=true;
            endif
        endif
step3   /* no more free partition to recombine */
        if cube_buddy_found=false and
        level_buddy_found=false then
            insert the cube address of BH(b_r,l_r,k_r,2) into the list
            of PART[F(b_r,k_r,l_r)]
        else
            goto start
end LBFR;
```

The time complexity of LBFR is identical to the time required by CBFR, since only step 1 and step 2 are interchanged in CBFR. Thus, LBFR takes $O(k)$.

Based on the different ways for allocation and recombining, four different partitioning strategies, combining all possible alternatives, are considered: CBFA-CBFR, CBFA-LBFR, LBFA-CBFR, and LBFA-LBFR. Performance metrics for partitioning are discussed next.

5-Fragmentation:

Like memory management, partitioning of parallel architectures suffers from both internal and external fragmentation. Allocation of partitions of predefined sizes to incoming tasks results in internal fragmentation. External fragmentation occurs when there is no partition large enough to accommodate the incoming task, though the cumulative size of all scattered available partitions is large enough. Both internal and external fragmentations reduce the resource utilization.

5.1 Internal fragmentation

As requests are of arbitrary sizes, it is not always possible to find a partition that matches in size an incoming request, therefore more processors are allocated. This results in internal fragmentation.

The internal fragmentation is fully determined by the network topology and the probability distribution of request sizes. For example, in the case of banyan-hypercubes and hypercubes, a request of size $33 \le n \le 48$, is always granted a (3,4)-partition and a 64-cube, respectively. This yields to a 64-n idle nodes in the case of hypercube and only 48-n in the case of banyan-hypercubes. Therefore, internal fragmentation in BH's is smaller than that of hypercubes. Several definitions of internal fragmentation have been considered. One measure is defined in [9] as:

$$F_{internal} = \frac{\sum_{i=1}^{m} p_i (w(i)-i)}{\sum_{i=1}^{m} p_i w(i)}$$

Where m is the maximum task size, p_i is the probability that the size of the task is i, and $w(i)$ is the allocated partition size.

5.2 External fragmentation

External fragmentation occurs when an incoming request cannot be satisfied. This situation is called an overflow. The measure of external fragmentation depends on several factors: the size distribution of the task, the lifetime of the tasks and the schemes used to allocate and recombine partitions in the system. Similar to the two dimensional buddy system [7], the external fragmentation is defined as the ratio of the total size of available partitions to the total system size, and measured when an overflow occurs. Formally, it is defined as follows:

$$F_{external}=1-\frac{1}{P.T}\cdot\sum_{i=1}^{n} w(i).t_i$$

where n is the number of the tasks in the system, $w(i)$ is the size of the partition allocated to task i, t_i is the lifetime of task i, P is the total size of the system and T is the total time needed by the n tasks to finish their executions. For our simulation we use the following formula (which is a good approximation of the one given above) to measure the external fragmentation [7]: $(E_1+E_2+.....+E_N)/N$, where E_i is the fraction of the unallocated processors when the ith overflow occurs, and N is the number of overflows .

5.3 Total fragmentation

The total fragmentation F_t, results from the unallocated processors in the internal fragmentation and the idle partitions in the system. It is a weighted sum of both internal fragmentation F_i and the external fragmentation F_e [9]: $F_t=(1-F_e).F_i+F_e$.

6-Simulation

We have simulated the various partitioning strategies on a BH of 1K nodes, using uniform and exponential distributions for request sizes, and uniform distribution for task lifetime. Since simulation results show that all four alternatives are similar, only CBFA-CBFR is presented in this paper for both uniform and exponential distributions. We have also simulated the buddy system (BS) strategy on a hypercube of 1K nodes, for comparison purposes. Internal, external and total fragmentation were measured.

Since the BH partitions do not need to be a power of 2 like the hypercube partitions, the internal fragmentation in BH is much smaller than that of hypercubes. Simulations confirm this and show that the internal fragmentation of the buddy system on the hypercube is 26% while that of TWPS on the BH is only 10%, figure 8 (a) and 9 (a).

However, the BS on hypercubes demonstrates a better external fragmentation over the TWPS on BH's. The BS on the hypercube has an external fragmentation of 21%, whereas TWPS has 50% on the average. But note that they become comparable for large request sizes, as shown in figure 8 (b) and 9 (b). Furthermore, it should be noted that the external fragmentation, unlike the internal fragmentation, depends on the sequence of incoming requests and their lifetime. Therefore, external fragmentation can be reduced in various ways, for example by using appropriate scheduling and partition compaction [1][2].

The total fragmentation, which reflects the overall performance, is plotted in figure 8 (c) and figure 9 (c) for uniform and exponential distributions, respectively. As can be seen, TWPS and BS become similar for large request sizes.

We have also simulated the effect of the network height h, on the internal fragmentation. We concluded that the internal fragmentation is reduced as the height gets larger. Figure 10 shows the internal fragmentation in a 4 K nodes network with h=2 and k=11 and with h=4 and k=10, using an exponential distribution.

7-Conclusion

We have developed various partitioning strategies and a data structure for BH networks, and studied their performance. Internal, external, and total fragmentation have been simulated. The simulation results show that TWPS has a better internal fragmentation than the BS on the hypercube. For large request sizes the total fragmentation of BS and TWPS become similar with a smaller internal fragmentation in BH's. We have also shown that the internal fragmentation is reduced for BH's with larger height. Future work includes the extension of these strategies to BH's with spread larger than two. In addition, these partitioning strategies will be rendered distributed to reduce partitioning overhead.

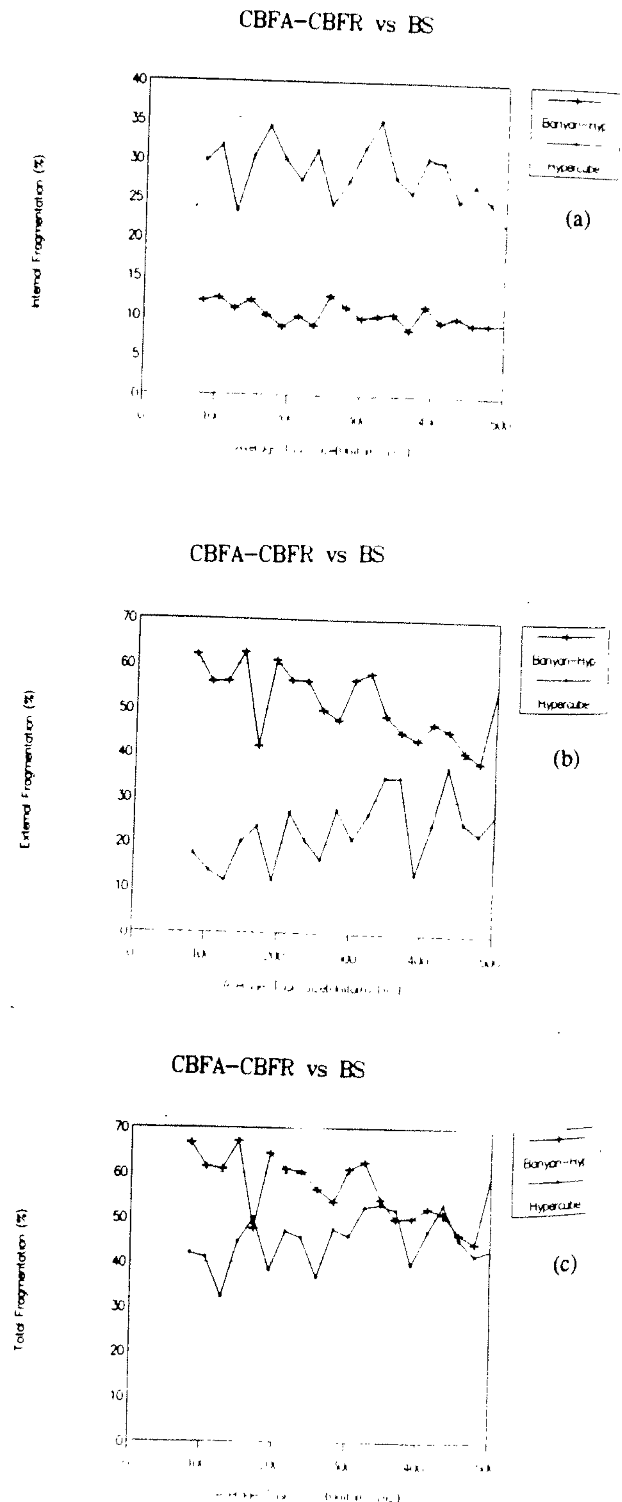
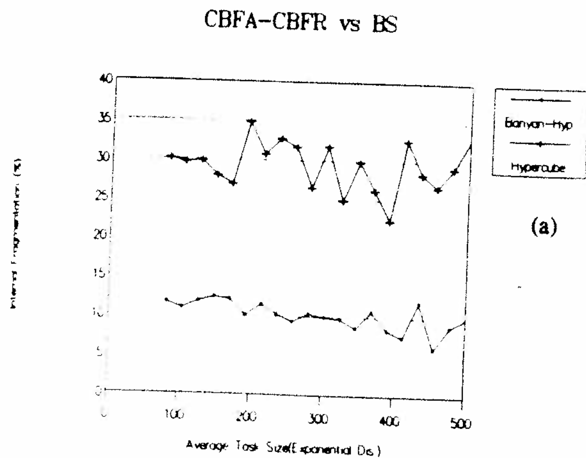
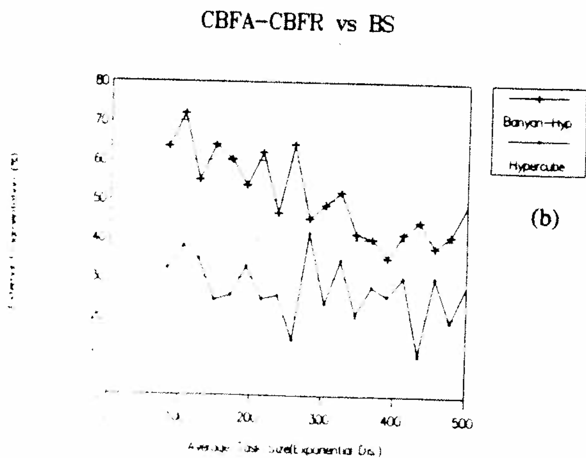


Figure 8





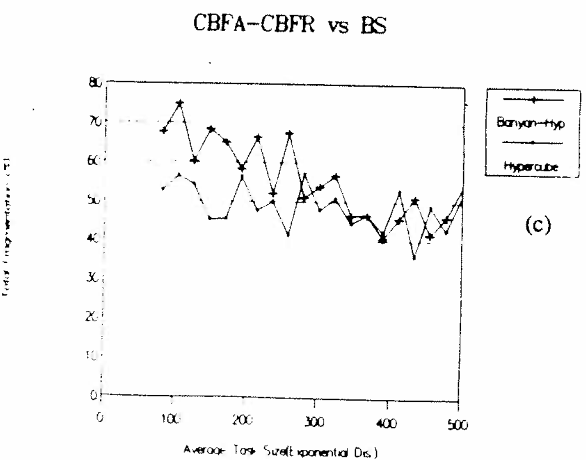


Figure 9

CBFA-CBFR

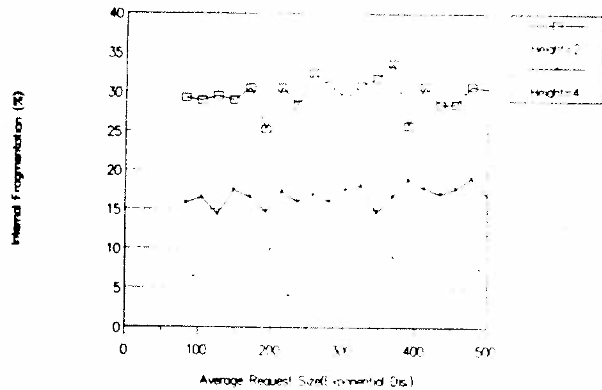


Figure 10

References

[1] G.I. Chen and T.H. Lai, "Virtual Subcubes and Job Migration in a Hypercube," *Int'l Conf. Par. Proc.*, pp. II73-II76, 1989.

[2] M. S. Chen and K. G. Shin, "Task Migration in Hypercube Multiprocessors," *ACM*, 1989, pp 105-111.

[3] 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, December 1987.

[4] R. L. Goke, "Banyan Networks for Partitioning Multiprocessor Systems", Doctoral Dissertation, University of Florida, 1976.

[5] W. D. Hillis, *The connection Machine*, MIT Press, 1985.

[6] M. Jeng and H.J Siegel, "A Distributed Management Scheme for Partitionable Parallel Computers," *IEEE Transactions on parallel and Distributed Processing*, vol. 1, pp. 120-126, Jan. 1990.

[7] K. Li and K. H. Cheng, "The Two Dimensional Buddy System for Dynamic Resource Allocation in The Partitionable Mesh Connected System," *Technical Report No. UH-CS-89--03*, Dept. of Computer Science, U. of Houston, 1989.

[8] NCUBE Corp., *NCUBE/ten: An Overview*, Beaverton, OR, Nov. 1985.

[9] J.L. Peterson and T.A Norman, *Buddy Systems*, *Communication of ACM*, June 1977, pp. 421-431

[10] C. L. Seitz, "The Cosmic Cube," *Commun. ACM*, vol. 7, pp. 22-33, 1985.

[11] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comput.*, vol. C-29, pp. 791-801, Sept. 1980.

[12] H.J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture*, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furth. Eds. Washigton, DC: IEEE Computer Society Press, 1987, pp. 387-407.

[13] A. Youssef, B. Narahari, "Banyan-Hypercube Networks," *IEEE Transactions on Parallel and Distributed* Systems, pp. 160-169, April 1990.

[14] A. Youssef, B. Narahari, "Topological Analysis of the Banyan-Hypercube Networks", submitted to the *Second IEEE Symposium on Parallel and Distributed Processing*.