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 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-hypercubes 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 of 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 banyan is an L-level banyan where all the nodes have the same number of nodes, which is S^L. Figure 1 shows a rectangular banyan of spread 2 and another of spread 4.
A banyan-hypercube, denoted BH(h,k,s), where h ≤ k + 1 and s is the 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 2^s - 1. Therefore, each node is labeled by a pair (L,X) where 0 ≤ L ≤ h - 1 and X is a cube address of the form x_{k+1}...x_{k+1} in the number system of base s. The total number of nodes in the network is h^s. 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 ≤ s ≤ h, constitute a graph which is isomorphic to BH(l,k,s). Therefore, any subnetwork of BH((k,s)) of l levels, starting from level b, such that the nodes in each level are interconnected in a k-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).

One can easily see that a partition BH(b_i,k+1,2) is still not uniquely identified. This is because for a given base b_i and a number of levels l, we might have 2^{k+1} k-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,k+1,2) of a BH(0,h,k,2), a representation based on the ternary representation in [3] is used to address the k-cube of each partition. This will be referred to as the cube address of the partition. The address of each k-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 (000,010,100,101) 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 \( \text{**} \) (i.e. \( \text{**} \)) and its subbanyan-hypercubes BH(0,2,1,2) have the following cube addresses, from left to right, 00*, 01*, 10*, 11*.

**Figure 3**

**Figure 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 a new 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.
The address of an available partition BH\((h,i,k,l,2)\) is then located in the list pointed to by \(\text{PART}(I(b,k,l,2))\). The space complexity of \(\text{PART}\) is now reduced to \(O(UO(\text{list\_space}))\), where \(U\) is given by proposition 1. To compute the term \(O(\text{list\_space})\) 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 \(k\)-cubes, using the buddy system, in a hypercube of dimension \(k\) is \(2^{k-k}\), 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 \(K_h(l,k)\) of \((l,h,k,2)\)-partitions having their top level coincide with the top level of the BH is:

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

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

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

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

\[
P_h(I,l,k) = \begin{cases} 
(k+l+2)2^{k-l} & \text{if } l \neq 1 \\
h2^{k-l} & \text{if } l = 1
\end{cases}
\]

**Proof:** Let \(P_h(k,l,I)\) and \(P_h(l,k,I)\) be the total number of \((l,h,k,2)\)-partitions in BH\((0,h,k,2)\) and in BH\((0,h-1,k,2)\), respectively. \(P_h(k,l,I)\) can be computed using the following recurrence relation:

\[
P_h(k,l,I) = P_h(k,l,I+1) + K_h(l,k)
\]

Where \(K_h(l,k)\) is the number of \((l,h,k,2)\)-partitions such that their top level coincide with the top level of BH\((0,h,k,2)\). Using lemma 2, we have:

\[
a_m = a_{m-1} + 2^{k-1}
\]

where \(a_m\) is the number of levels in the network. Two cases are considered:

**Case 1:** \(l=1\). In this case the value of \(K_h(l,k)\) is \(2^{k-1}\), and the recurrence becomes:

\[
a_m = a_{m-1} + 2^{k-1}
\]

The initial value of this relation is when \(m=1\), that is the hypercube case. This value is computed by lemma 1 and is \(a_0 = 2^{k-1}\). The solution of this relation is simple and the result is:

\[
a_m = 2^{k-1},
\]

and therefore \(K_h(l,k) = a_m = 2^{k-1}\).

**Case 2:** \(l \neq 1\). Lemma 1 introduces two sub-cases to compute the value of \(K_h(l,k)\) in BH\((0,h,k,2)\) when \(l \neq 1\):

\[
K_h(l,k) = \begin{cases} 
0 & \text{if } k < m-1 \\
2^{k-l} & \text{if } k \geq m-1
\end{cases}
\]

Therefore, two different recurrence relation result from the above conditions.
The initial value for these two relations can be derived by observing that the height of the smallest BH network which has $(l,k)$-partitions must be $2l$. Therefore, $m$ can take values between $l$ and $m$, where $k$ is the height of the network. The value of $a_k$ can be computed by applying lemma 2 to $BH(0,l,0,k,k)$, leading to $a_k = 2^{(k+1)}$, since $l = l = k = k$. From relation (2) we conclude that $a_{m+k} = (m+k)2^{(m-1)}$, for $m+k > 1$. In particular, $a_{m+k} = a_{m+k} = (m+k)2^{(m-1)}$. From relation (1) we have $a_k = a_k = a_k = a_k = a_k = a_k$. Therefore, $a_k = a_k = a_k = a_k = a_k = a_k$.

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 $l$ and $k$, the number of partition $P_{k} = (l,k)$ is large. For $l = l = k = k$, the total number of $(1,0)$-partition is computed by theorem 1, $P_{k} = (1,0,0) = 2$. 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 $b$, we may have $2^{b} - 1$ free processors in the worst case. In that case the space needed by the list $O_{list, space} = O(2^{b})$. Hence, the space complexity of PART is $O(2^{b})$ and that of $P$ is $O(hk2^{b-1}) = O(hkN)$, where $U$ is given in proposition 1 and $N = h$.

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 multiprogramming 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 banyan hysteresis system [13], and is possible only if the partition, $BH(b,b,b...b)$, is not a full banyan-hypercube, $b < k - 1$, 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.

4.1 Computation of the partition size

For an incoming task size $n$, the computation of the size of the required partition is such that the internal fragmentation is minimized. The logarithmic width $k$ and the height $h$ of the required partition are such that $n \leq 2^h$. For a given value of $n$ there may be more than one solution for $l$ and $k$. 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: $l = 1$ and $k = 6$, or $l = 2$ and $k = 5$, or $l = 4$ and $k = 4$. A preferable choice of $l$ and $k$ is such that $l$ is the smallest value among all possible values. In the case of $n = 57$, that choice is $l = 1$ and $k = 6$. This is because the total number of $(l,k)$-partitions is greater than the number of $(l,k)$-partitions of the same size, if $l < l$ and $k < k$.

**Theorem 2**: The total number of $(l,k)$-partitions is greater than the total number of $(l,k)$-partitions of the same size.

**Proof**: By theorem 1, the values of $p_l$ and $p_k$ are $2^{2l}$ and $(k-l)+2^{2l}$, respectively, since $l = l$ and $k = k$. Let us assume that $p_l < p_k$, then we have $2^{2l} < (k-l)+2^{2l}$. Since $k < k$, it follows that $2^{2l} < 2^{2l}$, and therefore $k < k$. Recall that $l$ is the top level of the partition and $k = k$, the highest possible base of $(l,k)$-partitions. So, $l$ should be greater than or equal to $k = k$, contradicting the fact that $l < l$.

**Case 2**: $l = 1$. The proof in this case is similar to case 1.
To compute \( l_i \) and \( k_i \) for a given \( n \), the procedure \textbf{FIND} and \textbf{FIND}-\textbf{SMALLEST} are given below. First the procedure \textbf{FIND} computes one possible value of \( l_i \) and \( k_i \), then the procedure \textbf{FIND}-\textbf{SMALLEST} finds the value with the smallest height.

**Procedure** \textbf{FIND}(\( n, l_i, k_i \))

1. \( m \) = integer.
2. \( m = 1 \).
3. while \( 2^m \leq n \) do
   4. \( i = m^2 \).
   5. if \( l_i \leq m+1 \) then
      6. \( k_i = m \).
      7. return;
   8. else
      9. \( m = m + 1 \);
 10. end if
11. endwhile
12. end FIND;

**Procedure** \textbf{FIND}-\textbf{SMALLEST}(\( l_i, k_i \))

1. \( p, q \) = integer.
2. \( p = 1, q = k_i + 1 \).
3. while \( (p < q) \) do
   4. if \( p^2 \leq (2^q) \) then
      5. \( i = p, k_i = q \).
      6. return;
   7. else
      8. \( p = p + 1 \);
      9. \( q = q + 1 \);
 10. end if
11. endwhile
12. end FIND-\textbf{SMALLEST};

The procedure \textbf{FIND} takes \( O(\log n) \), while \textbf{FIND-\textbf{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 \( O(\log n + h) \).

### 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 \( l_i \) and \( k_i \) are computed, the possible values of the base \( b_i \) of the \((l_i,k_i)\)-partitions are in the range from 0 to \( k_i-l_i+1 \) if \( l_i \neq 1 \), and in the range from 0 to \( h-1 \) if \( l_i = 1 \) (definition 1). For each value of \( b_i \), starting from the largest value, the TWPS allocates the first free partition in the list pointed to by the entry \textbf{PART}(\( b_i, l_i, k_i \)) if there is no available partition for any possible value of \( b_i \), 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 \( l_i = l_i \) and \( k_i = k_i \). If there is a free partition, then it is allocated; otherwise, for each possible value of \( l_i \), between \( l_i \) and \( k_i \), the logarithmic width \( k_i \) is incremented by 1 (up to \( k \)) searching for a \((l_i,k_i)\)-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** \textbf{CBFA}(\( b_i, l_i, k_i, l_{\text{max}} \))

1. \( l_i \) and \( k_i \) are the number of levels and the logarithmic width \( l_{\text{max}} \) of the partition of the incoming task.
2. \( l_i \) and \( k_i \) is found to be true if there is an available partition.
3. If a free partition \( BH \) is found, then it is returned.
4. \( l_i = l_i, \ k_i = k_i \)
5. found = true;
6. step 1: compute the set \( b \) of the possible base levels \( b_i \) of \((l_i,k_i)\)-partitions: \( 0 \leq b_i \leq k_i-l_i+1 \) if \( l_i \neq 1 \) and \( 0 < b_i \leq k_i \) if \( l_i = 1 \)
7. step 2: \( l_i \leq h \)
8. step 2.1: for each \( b_i \) in \( B \) do
9.   if there is an available partition \( BH(b_i,l_i,k_i) \) at the base level \( b_i \) then
10.      found = true and exit. /* exit while loop */
11.   endif
12. endfor
13. step 2.2: /* Look for a larger partition */
14.   for each base level \( b_i \) in \( B \) do
15.     check all the \((l_i,k_i)\)-partitions, where \( k_i < k_i \) \( \leq k \)
16.     if there is an available partition then
17.       found = true and \( k_i = k_i \).
18.     endif
19.   endfor
20. end if
21. endwhile
22. step 3: if found then
23.     \textbf{SPLIT} \((b_i,l_i,k_i)\);
24. else
25.     put request in the waiting queue and found = false
26. endif
27. end CBFA;

The performance of \textbf{CBFA} is dominated by step 2. 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 \( O(\log h) \) since finding an \((l_i,k_i)\)-partition takes \( O(\log k) \). Therefore, the overall time of \textbf{CBFA} is \( O(h \log h) = O(h^2 \log k) \). If we have a full Banyan Hypercube, \( h = k + 1 \), then the number of steps becomes \( O(k^2) \). If \( N \) is the number of processors, \( N = (k+1)^2 \), then \( k \) is equal to \( \log(N/(k+1)) \), henceforth, we have \( O(\log^2(N/k)) \).

**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 \textbf{CBFA}, \((l_i,k_i)\)-partitions are searched, for all possible values of \( b_i \), where initially \( l_i = l_i \) and \( k_i = k_i \). If there is a free \((l_i,k_i)\)-partition, then it is allocated; otherwise, for each possible value of \( k_i \), between \( k_i \) and \( k_i \), the number of levels is incremented by 1 (up to \( h \)) searching for a \((l_i,k_i)\)-partition. The search stops when a partition is found. If there is no free partition the request is queued.

The performance of \textbf{LBFA} is similar to the performance of \textbf{CBFA} procedure. Step 2.1 is the same in both procedures. Step 2.1 in \textbf{LBFA} takes \( O(h^2) \) since we are looking for an
(l,k)-partition with larger height. The total time is then O(k^2).
Similarly to the performance analysis of CBFA, for a full
banyan-hypercube the time becomes O(\log^2(N/k)).

Procedure LBFA(b_i,l_i,k_i,l_i,k_i)

/* l_i and k_i are the number of levels and the logarithmic width */
/* l_i of the partition of the incoming task. */
/* k_i found := flag set to true if there is an available partition */
/* If a free partition BH is found, then it is returned */
/* l_i=1 ; k_i=k_i */
found=false;
step 1: compute the set B of the possible base levels b_i of
(l_i,k_i)-partitions:0\leq b_i\leq k_i-1 if l_i=1 and
0< b_i \leq k_i-1 if l_i=2;
step 2: while (k_i\leq 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_i,k_i)-partitions, where
l_i\leq k_i+1 and b_i if the network is full; otherwise
l_i\leq k_i.
If there is an available partition then
found=true; l_i=l_i; 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 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_1...a_{k_i-1}...a_{k_i} = \ldots) is found either by CBFA
or by LBFA the splitting algorithm compares the size of BH_i
with the size of the required (l_i,k_i)-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_i;
and denoted BH_i. The other resulting partitions are returned to
the data structure, using the RECOMBINE procedure in the
algorithm below. If the logarithmic width of BH_i is equal to k_i
then BH_i is allocated: otherwise it is split vertically k_i times,
where l_i\leq k_i. All the resulting partitions will be returned to
the data structure except for one partition, the one with cube address
A_1=a_1...a_{k_i}...a_{k_i} = \ldots, 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 recombinating 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(k) 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(k)+O(k), since k_i=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)
/* BH_i is large and l_i > i, then let b_i be the base of the */
/* required partition. The value of b_i is such that b_i>2b_i and */
/* b_i+1=stop_i, where stop_i=b_i+1 is the top level of the */
/* allocated partition. Note that b_i + stop_i = stop_i is the top level */
/* of the required partition. */
/* top_i , top_i , b_i , integer */
/* top_i = b_i+1 */
/* top_i = stop_i */
step 1: if l_i=1, then
allocate the partition BH(b_i,l_i,k_i,2) and stop
endif
step 2: if l_i=l_i, then
Find the largest value of b_i such that b_i\leq b_i \leq stop_i-1
and top_i=b_i+1-1stop_i.
if top_i > stop_i then /* recombine upper level-buddy */
RECOMBINE(top_i+1 ,stop_i ,k_i)
endif
if b_i>stop_i then /* recombine lower level-buddy */
RECOMBINE(b_i,b_i-b_i,k_i)
endif
step 3: if k_i=k_i, then
split vertically, using the buddy system, until k_i=k_i
endif
step 4: allocate BH(b_i,l_i,k_i) to the incoming task.
end SPLIT;

4.4 Recombining algorithm

The third step of our partitioning system consists of recombinating 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 recombinating policies are considered.

Cube-buddy first_recombining (CBFR):
Recombinating is repeatedly done, first on all the free cube-
buddy partitions then on the upper and lower level-buddy
partitions. Figure 7 (a) when the cube-buddy first
recombining is better. The recombinating 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 1**, 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.

(a) Free partitions
(b) Released Partition

Figure 7
procedure CBFReq(b1,l,k):
  /* Let BH be the partition to be recombed */
  l=0, w=b1, b=b0;
  cube_buddy_found, level_buddy_found: boolean=false;
  start:
  l=0, w=b1, b=b0;
  step1
  if there is a cube address of the cube-buddy partition of BH, in the list pointed by PART[F(b1,l,k),] then
    k=k+1; cube_buddy_found=true;
  endif
  step2
  if the upper level-buddy of BH, is in the list pointed to by PART[F(b1,l,k),], where b=b+b1, and
    l<s+b1-1-l (1<s+l if BH is not full) then
    l=ll; level_buddy_found=true;
  endif
  if the lower level-buddy of BH, is in the list pointed by PART[F(b1,l,k),], where b=b-1, and
    l=b1-1-b then
    b=b1; l=w1;
  endif
  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, the first step (i.e. 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<s+1.

Level_buddy_first_recombing (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 recombinimg is better. The recombinimg 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 recombinimg 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 it first start recombinimg with the free level_buddy partition (i.e. interchange step 1 and step 2 in CBFR).

procedure LBFR(b1,l,k):
  /* Let BH be the partition to be recombed */
  l=0, w=b1, b=b0;
  cube_buddy_found, level_buddy_found: boolean=false;
  start:
  l=0, w=b1, b=b0;
  step1
  if the upper level-buddy partition of BH, is in the list pointed to by PART[F(b1,l,k),], where b=b+b1, and
    l<s+b1-1-l (1<s+l if BH is not full) then
    l=ll; level_buddy_found=true;
  endif
  if the lower level-buddy partition of BH, is in the list pointed to by PART[F(b1,l,k),], where b=b-1, and
    l=b1-1-b then
    b=b1; l=w1; level_buddy_found=true;
  endif
  else
    b=min(b1,b);
  endif
  else if cube_buddy_found=false then
    if the cube-buddy partition of BH, is in the list
      pointed by PART[F(b1,k,k),] then
        k=k+1; cube_buddy_found=true;
      endif
    step3 /* no more free partition to recombe */
    if cube_buddy_found=false and
      level_buddy_found=false then
      insert the cube address of BH(b1,l,k,2) into the list of
      PART[F(b1,l,k)]
    else
      goto start
    end
  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 recombinimg, four different partitioning strategies, combining all possible alternatives, are considered: CBFA-CBFR, CBFA-LBFR, CBFR-CBFA, 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 can 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:

\[ F_{internal} = \sum_{i=1}^{m} p(w(i)) \]

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 recombin 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:
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]: 
\[
\left( E_1 + E_2 + \ldots + E_N \right) / N,
\]
where \( E_i \) is the fraction of the unallocated processors when the \( i \)th 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]: 
\[
F_t = (1 - F_i) F_i + F_e.
\]

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 BHs. The BS on the hypercube has an external fragmentation of 21%, whereas TWPS has 30% 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.
References


