PROCEEDINGS

OF THE

FOURTH ANNUAL PARALLEL PROCESSING SYMPOSIUM

APRIL 4-6, 1990

EDITED BY
LARRY H. CANTER

Co-Sponsored by

THE ORANGE COUNTY IEEE COMPUTER SOCIETY

CALIFORNIA STATE UNIVERSITY, FULLERTON

FULLERTON, CALIFORNIA

Implementing Priority Queue on Hpercube Machine

M. R. Meybodi
Computer Science Department
Ohio University
Athens, Ohio 47501

August 1989

Abstract

To efficiently implement parallel algorithms on parallel computers, concurrent data structures (data structures which are simultaneously updatable) are needed. In this paper we derive concurrent data structures for implementing priority queues. We show how priority queue operations can be implemented on distributed-memory message-passing multiprocessors with hpercube topology. Three different implementations will be presented. The first implementation is based on heap data structure. The heap is mapped into a linear chain of processors which is then embedded into the heprcube. The other two implementations are based on banyan heap data structure. They differ in the way that the banyan heap is mapped into the hypercube. The key advantage of banyan heap over the heap is that with banyan heap it is possible to retrieve elements at different percentile levels.

Keywords and Phrases: Concurrent Data Structure, Hpercube, Banyan Heap, Parallel Algorithm

1 Introduction

Priority queue data structure is a very important data structure which has found application in varieties of situations such as: discret event simulation systems, timed-shared computing system, finding shortest paths in a graph [17], finding the minimum spanning tree of a graph [17], and iteration in numerical schemes based on the idea of repeated selection of an item with smallest test criteria [16], to mention a few.

Such a data structure is a set of elements each of which has an associated number, its priority for it. For each element $x, p(x)$, the priority of x is a number from some linearly ordered set. Standard operations on a priority queue are *INSERT*, which inserts an element and its associated priority into the priority queue, and *XMAX*, which deletes the element with the highest priority from the queue. Let P denote the set of all element- priority pairs. Define

$$P(s) = \{(x, p(x)) | p(x) = s \text{ and } (x, p(x)) \in P\};$$

The effect of priority queue operations are as follows:

INSERT$(x, p(x))$:

$P \leftarrow P \bigcup \{(x, p(x))\}$.
Response is null.

XMAX:

$P \leftarrow P - P(p_{max})$ where $P(p_{max})$ is the pair with the highest priority.
Response is $P(p_{max})$.

Since most of the applications of priority queues are computationally intensive, it is important to have an efficient implementation. Three kinds of implementations of priority queue have been reported in the literatures: sequential algorithms for implementation on uniprocessor. Parallel algorithmic architectures for realization on hardware, and parallel algorithms for implementation on parallel computers.

There are number of implementations for priority queue on a uniprocessor. Priority queue can be maintained as a sorted list, in which case, deleting the element with the highest priority takes $O(1)$ time but insertion takes $O(N)$ time, or it can be maintained as an unsorted list, in which case insertion takes constant time but $XMAX$ takes $O(N)$ time. N is the number of element-priority pairs in the priority queue. Other implementations which lend themselves to logarithmic time are height or weight balanced tree, partially ordered tree(heap), leftist trees suggested by C. A. Crane [30], pagodas [37], skew heaps [38], implicit heaps [30], and binomial queues [39] . The most widely used implementation of priority queue is with a heap data structure. A heap is a full k-array tree such that the priority of the element at any node in the tree is greater than priority of the element at each of its children. (Note that the definition of heap given here is slightly different from) Thus the element with the highest priority is always at the root of the heap. Operation *INSERT* is performed by adding the pair to the last level of the heap and then converting the resulting complete tree into a heap by pushing that pair up the tree recursively. $XMAX$ works by replacing the pair at root with the pair residing in the rightmost node of the last level of of heap and then converting the resulting complete tree into a heap by pushing the pair at the root down the tree recursively. If $k = 2$ then we have binary heap which is the most wildly studied special case.

A number of multiprocessor design for maintaining priority queue have been proposed in the literatures. These design can be classified into two main groups. 1) designs with small number of processors each having an small amount of memory, and 2) designs with small number of processors each having a large amount of memory. Group 1 designs [13,22-27,33] which in turn can be classified into systolic tree designs and systolic array designs are suitable for implementation by VLSI hardware whereas group two designs [2,11,29] can be implemented either in VLSI or any general purpose tightly or loosely coupled multiprocessor architecture.

Existing algorithms for multiprocessors are divided into two groups: those for shared memory multiprocessors and those for distributed memory mutiprocessors. When designing algorithm for shared memory multiprocessors the focus is on reducing the interference between concurrent processes accessing the data structure whereas in the case of distributed memory multiprocessor the focus is on exploiting the parallelism in the data structure. Concurrent algorithms for manipulating binary tree due to Kung and Lehman [18] , concurrent algorithms for B-tree due to Lehman and Yao [19], algorithms for concurrent search and insertion of data in AVL-tree and 2-3 trees due to Ellis [20,21], concurrent algorithm for insertion and deletion on the heap due to Rao and Kumar [8], and Biswas and Browne [1] are examples of algorithms for shared-memory multiprocessor. Examples of algorithms for distributed memory multiprocessor are: balanced cube due to Dally [9] and sorted chain due to Omondi and Brock. Both of these algorithms are for dictionary data structure for implementation on hypercube multicomputer.

It should be noted that most of the machines or algorithms cited in this paper offer capabilities which go beyond priority queue operations. Other operations supported by most of the cited algorithms or machines are operations SEARCH, NEAR, UPDATE, and DELETE on set of key-record pairs. In the context of priority queue a record is the element and the corresponding key is the priority associated to that element.

In this paper we present three implementations of priority queue on distributed-memory message passing multiprocessor with hpercube topology. The first implementation is based on heap data structure. The heap is mapped into a linear chain of processors which is then embedded into hypercube. The other two implementations are based on banyan heap data structure. They differ in the way that the banyan is mapped into the hypercube. The key advantage of banyan heap over the heap is that with banyan heap it is possible to retrieve elements at different percentile levels. A hpercube computer is briefly described below.

An d dimensional hpercube machine consists of $2^d = n$ processor nodes interconnected to one another to form an d dimensional cube. In an d dimensional cube two nodes are connected if and only if the binary representation of their numbers differ by one and only one bit. Each processor node in the hpercube has a processor and a local memory for that processor. The processors work independently and asynchronously and communicate by passing messages [3]. Examples of 1 , 2 , 3, and 4 dimensional Hpercube are shown in figure 1.

The rest of this paper is organized as follows. Section 2 gives a heap based implementation of priority queue on a distributed-memory message-passing. Section 3 defines banyan graphs and banyan heaps. In section 4 we discuss two different implementations for banyan heap and derive analytical formula for the percentile level of a retrieved element. The last section is the conclusion.

2 First Implementation

This Implementation is based on heap data structure. The heap is first mapped into a linear chain of n processors. This chain is then embedded into the hypercube in such a way

as to preserve the proximity property, i.e., so that any two adjacent processors in the chain are mapped into neighbor nodes in the hpercube [3]. The use of Gray codes is one known technique to obtain a mapping which preserve the proximity property. This technique can be explained as follows: If the binary representation of the node number in the chain is $b_{d-1}b_{d-2}...b_0$, then it is mapped into the node in the hpercube whose number , $c_{d-1}c_{d-2}...c_0$, is determined by the following equations.

$$c_i = b_i + b_{i+1}, \text{ if } (i < d-1)$$
$$c_i = b_i, \text{ if } i = d-1$$

For example, node number 26 (011010) in the chain is mapped into node 23(010111) in the hpercube. If $2^d < \log N$ then more than one level of the heap may be assigned to a single processor in the chain. This extension will not be discussed here.

Processor p_i , $(0 \leq i \leq n-1)$, of the chain stores the i^th level of the heap. Each node has 6 fields: DATA, PRIORITY, LCHILD, RCHILD,LEMPTYNODES, and REMPTYNODES. For a node, the DATA field holds an element and the PRIORITY field holds the priority associated with that element, LCHILD and RCHILD holds respectively pointers to the leftchild and rightchild of that node, and LEMPTYNODES and REMPTYNODES hold the number of null nodes (nodes with no information) in the left and right subtrees of that node. Initially, the DATA field of all the nodes are set to null and the PRIORITY field of all the nodes are set to -1. The LEMPTYNODES and REMPTYNODES fields of all the nodes at level i are initialized to $2^{n-i} - 1$. These two fields are updated as data elements are inserted into or deleted from the heap. Information about the number of empty nodes is used by *INSERT* operation to decide which path in the heap should be followed during the insertion process. Lack of such information may lead to an overflow situation in the last processor. This happens if the *INSERT* operation moves along a path in which all the nodes are non-empty. Now we describe operations XMAX and INSERT.

The process running on each processor in the hypercube consists of two distinct parts. The first part belongs to the application that runs on all the processors of the hpercube. The second part, called executive, is responsible for the execution of the priority queue operations. We use E_i to denote the resident executive of processor p_i. An executive can receive and process many messages simultaneously. Priority queue operations are initiated by the application part of the processes running on the processors of the hpercube. An operation issued by a process is communicated to the executive of that process. The executive then send that operation to the processor at the head of the chain (p_0) for execution. The priority queue operations received by processor p_0 will be executed in a piplined fashion along the chain of processors. The response to the XMAX operation produced by p_0 is forwarded to the processor which originally initiated the operation. An executive receives instruction either from another executive or from the application part of the process to which it belongs.

The algorithms for *XMAX* and *INSERT* are different from their sequential counterparts in that they both perform the restructuring process from the top. The following algorithm

explains the INSERT operation when insertion is performed from the top. This new INSERT operation traverse the heap from the top to perform the restructuring process and unlike conventional algorithm it does not neccessarily expand the heap level by level.

XMAX operation: When executive E_0 receives operation *XMAX* from another executive or from the application part of its own process it generates an operation called *ADJUST* which propagates through the chain of processors and converts the binary tree (after the root is removed) into a heap. With respect to operation *XMAX*, all the processors except processor p_1 perform the same set of actions. We describe operation *XMAX* as follows:

processor p_1: When processor p_0 receives operation *XMAX*, it reports the element with the highest priority to the processor that issued the XMAX operation and then sends operation ADJUST to processor p_1.

processor p_i, $(0 < i < n)$: Let p be the address of the node in the local memory of processor p_{i-1} whose value had been moved up or reported by processor p_{i-1} in the previous step. On receiving operation *adjust*(p) by p_i, it finds the child of node p, q, that contains the element with higher priority, sends DATA(q) to p_{i-1} to replace DATA(p), and next sends *adjust*(q) to processor p_{i+1}. Processor p_{i-1}, $(i > 1)$, after filling up the node p with DATA(q), increases the LEMPTYNODES or REMPTYNODES field of node p by one depending on whether q has been the address of left or right child of node p. If both children of node p are empty then processor p_i will not generate any more *adjust* operation; it only sends a message to processor p_{i-1} asking to empty node p. No processor will receive an adjust operation until the most recent *adjust* operation issued by that processor is completely executed by its neighboring processor.

INSERT operation: When processor p_i , $(0 \leq i \leq n)$, receives operation *INSERT*(p,item) it performs the following actions. It first compares the priority of DATA(p) with the priority of *item*, if the priority of *item* is greater than that of DATA(p) it replaces DATA(p) by *item* and then issues *INSERT*(q,DATA(p)) to processor p_{i+1}. If the priority of item is less than that of DATA(p), then processor p_i only sends *INSERT*(q,item) to p_{i+1} . Q is the address of the right child of node p if LEMPTYNODES(p)< REMPTYNODES(p) and is the address of the left child of node p if LEMPTYNODES(p) > REMPTYNODES(p). Processor p_i also decreases the LEMPTYNODES(p) or REMPTYNODES(p) by one, depending on whether the item will be inserted into the right (q=RCHILD(p)) or left (q=LCHILD(p)) subtree of node p. *INSERT* operation will not be propagated further down if node p is a null node which in that case the only action performed by processor p_i is to store the element in DATA(p).

The response time for *XMAX* is $O(\log N)$. Because it takes $O(\log N)$ time for *XMAX* operation to reach processor p_0 and it also takes $O(\log N)$ time to send the result produced at processor p_0 back to the processor which initiated the operation. The pipeline period for both the XMAX and *INSERT* operations is $O(1)$, independent of the length of the chain of processors. The execution time for each of the *XMAX* or *INSERT* operation is $O(\log N)$.

This is due to the fact that it may take $O(\log N)$ time for a new element to find its correct place within the heap or to fill up the gap produced as a result of $XMAX$ operation.

Remark 1 Information stored in the LEMPTYNODES and REMPTYNODES fields of the nodes are used by INSERT operation to decide which path in the heap should be followed during the insertion process. This unique path can be also computed on the fly by the INSERT operation [5,8]. This requires the system to maintain two pieces of information: *last*, the index of the last nonempty node of the heap and *first*, the index of the leftmost node in the deepest level of the heap which contains at least one nonempty node. Let $I = last - first$ and $p = \log last$. Number I can be expressed as a p bit binary number. The binary representation of of I tells us whether to go to the right or left when we travel down the heap. At level i we use the i^{th} bit of I from the left to decide whether to go to the right (if 1) or left (if 0). Root of the heap is at level 1. Figure 2 illustrate the path followed by the INSERT operation and its corresponding bit pattern when $last = 13$ and $first = 8$. This procedure can be extended to apply to a k-ary heap where $k = 2^h$. If $I = last - first$ and $p = \log last$. Then at level l of the heap we use $(l+1)^{th}$ bits of binary representation of I to choose the node at the next level of the heap. The advantage of this procedure over the one described above is that it requires less memory space and also it expands the heap level by level just as in the conventional sequential algorithm. The usefulness of information about the number of empty nodes of the left and right subtree of the nodes become apparent when we talk about banyan heap in the following sections.

3 Banyan graphs and banyan heaps

A banyan graph is a Hasse diagram [34] of a partial ordering in which there is only one path from any base to any apex. A base is defined as any vertex with no arcs incident out of it and an apex is defined as any vertex with no arcs incident into it. A vertex that is neither an apex nor a base vertex is called an intermediate vertex.

An L-level banyan is a banyan in which the path from base to apex(or apex to base) is of length L. Therefore, in an L-level banyan, there are $L+1$ level of nodes and L-level of edges. By convention, apexes are considered to be at level 0 and bases at level L. In a banyan graph, the outdegree and the indegree of a node are called spread and fanout of that node. If there is an edge between two nodes, x at level i and y at level $i+1$, then we say x is the parent of y, and y is the child of x.

Definition 1 *A banyan is called a uniform banyan if all the nodes within the same level have identical spread and fanout.*

In a uniform banyan , the fanout and spread values may be characterized by L component vectors, $F = (f_0, f_1, ..., f_{L-1})$ and $S = (s_1, s_2, ..., s_L)$, the fanout vector and spread vector, respectively, where s_i and f_i denotes the spread and fanout of a node at level i.

Definition 2 *If $s_{i+1} = f_i$, $(0 \leq i \leq L-1)$, that is $F = S$, then the banyan is called rectangular. If $s_{i+1} \neq s_i$ for some i, then the banyan is non-rectangular.*

Definition 3 *A banyan is said to be regular if* $s_i = s$, $(1 \leq i \leq L)$, *and* $f_i = f$, $(0 \leq i \leq L-1)$, *for some constant* s *and* f. *Otherwise it is said to be irregular.*

Definition 4 *A banyan is an SW banyan if it has the following two additional properties:*
a) Two nodes at an intermediate level i, *have either no or all common parents at level* $i-1$.
b) two nodes at intermediate level i *have either no or all common children at level* $i+1$.

Definition 5 *An SW-banyan is said to be rectangular if it is regular and* $s_i = d$, $(1 \leq i \leq L)$, *and* $f_i = d$, $(1 \leq i \leq L-1)$, *for some constant* d.

Having introduced the notation and definitions we define Banyan heap.

Definition 6 *An L-level banyan heap is an L-level banyan such that the priority of the element at each node is equal or greater than the priorities of the elements at each of its children.*

Figure 3 shows an example of 4-level rectangular banyan heap. In this report we study the implementation of MxM rectangular SW-banyan heap with $d = 2$ on a n-cube where $2^n = \log M + 1$. M is the number of apexes. The restriction to MxM rectangular SW- banyan is in the interest of simplicity of presentation. In such banyans the number of levels is $\log M + 1$. Each node in the banyan heap has six fields: DATA, PRIORITY, LCHILD, RCHILD, LEMPTYNODES, and REMPTYNODES. In addition to the above six fields, each apex has another field called NEXT. This field is used to link apexes together. Initially, the DATA field of all the nodes are set to null and the priority field of all the nodes are set to -1. To initialize these fields, first partition the heap into N disjoint binary trees and then use the same rule as for the first design to initialize the LEMPTYNODES and REMPTYNODES fields of the nodes in each partition. The partitioning process starts with the leftmost apex and continues in increasing order of the apex numbers. The leftmost apex is numbered 1. Partition i is the set of all nodes which are reachable from apex i by moving down the heap and are not part of partition $i-1$. The root of the binary tree in partition i is apex i. The depth of a partition is the depth of the corresponding binary tree. The set of partitions and the initial settings of LEMPTYNODES and REMPTYNODES fields for an 8x8 SW-banyan is given in figure 4.

Definition 7 *An L-level partitioned banyan heap is an L-level banyan such that each partition of the banyan (as defined above) is a binary heap.*

Definition 8 *A partitioned L-level banyan heap is said to be full up to apex d if all the nodes in partitions* j, $(j < d)$, *are non-null and the nodes in the remaining partitions are null.*

Definition 9 *A node in a banyan heap is said to be reachable by partition from apex i if its parent is reachable by partition from apex i. Node l at level j+1 is reachable by partition from node k at level j if node l either has non-zero REMPTYNODES and it is the right child of node l,or has non-zero LEMPTYNODES and it is the left child of node l. An apex is reachable by partition from itself.*

Remark 2 The null nodes which are reachable by partition from a given apex will be filled up by insertions initiated at that apex unless the reachabilty of the nodes will change by a later deletion operation initiated at some other apexes. Reachability does not imply reachability by partition.

4 Second Implementation

The effective implementation of banyan heap on hpercube requires efficient mapping of the banyan structure among the processors of the hpercube. we examine two such mappings. The first mapping is obtained by first mapping the banyan into a linear chain and then embed the chain into the cube. See figure 5 . The second mapping is obtained by collapsing columns of the banyan into single processors [34]. Each base-apex path in the banyan with identically labeled vertices is mapped into one processor in the hypercube with the same label. According to this mapping if two nodes are adjacent in the banyan then they are mapped into two adjacent processors in the hpercube, those having labels that differs exactly in the i^{th} digit. See figure 6 for an example. In this section we consider the first mapping.

Before we give a detailed description of the operations XMAX and INSERT we show how to compute the addresses of the children of a given node. The nodes at a given level are stored sequentially in an array of size M in the local memory of the corresponding processor. Let n_{li} to denote the i^{th} node on the l^{th} level of the banyan where $0 =< i <= M, 0 <= l <= (logM + 1)$. Then n_{li} is connected to two nodes $n_{(l+1)i}$ and $n_{(l+1)m}$, where m is the integer found by inverting the i^{th} most significant bit in the binary representation of i. We call nodes $n_{(l+1)i}$ and $n_{(l+1)m}$ the right and left children of node n_{li}, respectively. Therefore, the left child and right child of node i at level k are stored respectively in the i^{th} location and m^{th} location of the array residing in the local memory of processor $k + 1$.

All the XMAX an INSERT operations initiated at different processors are sent to the head of the embedded chain and executed in a pipelined manner. When operation INSERT is received by processor p_0, it first finds the leftmost partition which has at least one empty node. This can be done using information stored in the REMPTYNODES and LEMPTYNODES fields of the apexes in $O(M)$ time. It then pushes the element requested to be inserted down the banyan heap using operation *insert-adjust*. The operation *insert-adjust* pushes the element down (along the paths from the root of the partition to the bases) until it finds its correct position. This needs $O(\log M)$ time. Thus INSERT is an $O(M)$ operation.

In the codes given below we use the following syntax and semantic for send and receive instructions. The instruction Send(<processor>,'<instruction>') sends instruction <instruction> to processor <processor> for execution. The execution of receive(<processor>,(informatio causes the information specified by the second argument be obtained from processor <processor> and forwarded to the requesting processor (the processor executing the receive instruction). A receive instruction executed by processor p_i is not complete until a message is received from processor p_{i+1}.

Upon receiving *INSERT*(p,(item,priority)) by processor p_0, it executes the following codes. P is the address of the leftmost apex, and (item, priority) is the pair requested to be inserted.

```
found ← false
While (not found ) do
  if DATA(p) ≠ null then
    if priority > PRIORITY(p)
      begin
        if LEMPTYNODES(p)≠ 0 or REMPTYNODES(p)≠ 0 then
          begin
            if LEMPTYNODES(p) > REMPTYNODES(p) then
              begin
                p'← RCHILD(p);
                REMPTYNODES(p) ← REMPTYNODES(p) −1
              end
            else
              begin
                p'← LCHILD(p);
                LEMPTYNODES(p)← LEMPTYNODES(p) −1
              end
            send(P2 , 'insert-adjust(p', DATA(p)));
            DATA(p) ←item; PRIORITY(p)← priority;
            found ← true
          end
        else
          p ← NEXT(p)
      end
    else
      begin ( priority < PRIORITY(p) )
        if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p)≠ 0 then
          begin
            if LEMPTYNODES(p) > REMPTYNODES(p) then
              begin
                p'← LCHILD(p);

                LEMPTYNODES(p) ← LEMPTYNODES(p) −1
              end
            else
              begin
                p' ← RCHILD(p);
                REMPTYNODES(p)← REMPTYNODES(p) −1
              end
```

to insert the elements in such a way that the element with the highest priority is always available at the leftmost apex in which case locating the correct apex to initiate the insertion takes O(M) time. This method seems to be more efficient due to the fact a portion of the time spent to find the correct position can be overlapped with the time spent to locate an apex with zero REMPTYNODES or zero LEMPTYNODES. In the algorithms presented above we have used the first approach. The second approach will be reported in another report.

From the properties of SW-banyan graphs and the above algorithms, we can state the following results.

Theorem 1 *Operation XMAX requires $O(M)$ time to complete.*

Theorem 2 *Operation INSERT requires $O(M)$ time to complete.*

Theorem 3 *The second implementation offers $O(M/\log M)$ throughput.*

Lemma 1 *a) The insert-adjust operation never encounters a node which is non-null and has zero LEMPTYNODES and zero REMPTYNODES. b) The insert-adjust operation always finds a null node to insert its element.*

Proof a) If operation xmax-adjust encountered a non- null node with LEMPTYNODES and REMPTYNODES fields equal to zero then it must have been initiated from an apex with both REMPTYNODES and LEMPTYNODES equal to zero. This is impossible according to the algorithm for INSERT. b) Proof is immediate from the proof of part a and the definitions of LEMPTYNODES and REMPTYNODES.

Remark 4 Deletion of an element from partition i may cause one of the elements in other partitions whose nodes are reachable from apex i to become null. This happens if a delete operation causes the xmax-adjust, on its way down the heap, to move up the content of one of the leaf nodes of partition i to fill up its parent which has been emptied by xmax- adjust operation at the previous step. The emptiness of this node now will be reflected in the REMPTYNODES or LEMPTYNODES of apex i. This node is now reachable by partition from apex i and will be filled by an insertion initiated at apex i. The maximum number of nodes that may become reachable by partition from apex i as a result of a deletion is equal to (L+1) - L', where L' is the depth of partition i.

Lemma 2 *Zero REMPTYNODES and zero REMPTYNODES for an apex does not imply that all the nodes in the corresponding partition are non- null.*

Proof From remark 3.

Lemma 3 *Apex i, $(1 \leq i \leq N)$, always contains the element which has the highest priority among the elements stored in the nodes of partition i.*

```
                send(p2, 'insert-adjust(p', (item,priority))');
                found ← true
             end
          else
             p ← NEXT(p)
       else
          begin
             DATA(p) ← item;
             PRIORITY(p) ← priority
          end;
```

Processor P_i, $(2 \leq i \leq L)$, upon receiving *insert-adjust*(p,(item,priority)) executes the following codes.

```
if DATA(p) ≠ null then
   if priority > PRIORITY(p) then
      begin
         if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p) ≠ 0 then
            begin
               if LEMPTYNODES(p) > REMPTYNODES(p) then
                  begin
                     p'← LCHILD(p) ;
                     LEMPTYNODES(p) ←LEMPTYNODES(p)−1
                  end
               else
                  begin
                     p' ← RCHILD(p);
                     REMPTYNODES(p) ← LEMPTYNODES(p) −1
                  end
            end
            send(p_{i+1},'insert-adjust(p',(DATA(p),PRIORITY(p));
            DATA(p) ←item; PRIORITY(p) ← priority
      end
   else
      begin
         if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p) ≠ 0 then
            begin
               if LEMPTYNODES(p) > REMPTYNODES(p) then
                  begin
                     p' ← RCHILD(p);
                     REMPTYNODES(p) ← REMPTYNODES(p) −1
                  end
```

```
                else
                   begin
                      p'← LCHILD(p);

                   LEMPTYNODES(p) ← LEMPTYNODES(p) −1
                end
                send(p_{i+1}, 'insert-adjust(p',(item,priority))');
                end
          else
             begin
                DATA(p) ← item; PRIORITY(p) ← priority
             end
    else
       begin
          DATA(p) ← item;
          PRIORITY(p) ← priority
       end;
```

XMAX operation first locates the apex which contains the element with the highest priority, reports that element to the outside world, and then fills up that apex with the element in one of its children. *xmax-adjust* is responsible for restructuring the banyan as it moves down the heap. When *XMAX*(p) is received by processor p_1, it executes the following codes. *P* is the address of the leftmost apex. This address is known to the outside world (front end computer).

```
p' ← p
while NEXT(p) ≠ nil and DATA(NEXT(p)) ≠ null do
   begin
      if PRIORITY(p) > PRIORITY(NEXT(p)) then p' ← p
      p ← NEXT(p)
   end
   send('outside world', DATA(p'));
   receive (p_2, ((PRIORITY(RCHILD(p')),DATA(RCHILD(p'))),
                        (PRIORITY(LCHILD(p')),DATA(LCHILD(p')))) )
   if DATA(RCHILD(p')) ≠ null or DATA(LCHILD(p'))≠ null then
      if DATA(RCHILD(p')) > DATA(LCHILD(p')) then
         begin
            DATA(p') ← DATA(RCHILD(p'));
            PRIORITY(p') ← PRIORITY(RCHILD(p'));
            REMPTYNODES(p') ← REMPTYNODES(p') + 1;
            send(p_2, 'xmax-adjust(RCHILD(p')')
```

```
        end
      else
        begin
          DATA(p') ← DATA(LCHILD(p'));
          PRIORITY(p') ← PRIORITY(LCHILD(p'));
          LEMPTYNODES(p') ← LEMPTYNODES(p') + 1;
          send (p_2, 'xmax-adjust(LCHILD(p')')
        end
    else
      begin
        DATA(p') ← null;
        PRIORITY(p') ← -1
      end
```

Processor p_i, $(2 \leq i \leq L)$, upon receiving *xmax- adjust*(p)) executes the following codes.

```
receive(p_{i+1}, ((PRIORITY(RCHILD(p)),DATA(RCHILD(p))),
                  (PRIORITY(LCHILD(p)),DATA(LCHILD(p)))) );
if DATA(RCHILD(p)) ≠ null or DATA(LCHILD(p)) ≠ null then
  if DATA(RCHILD(p)) > DATA(LCHILD(p)) then
    begin
      DATA(p) ← DATA(RCHILD(p));
      PRIORITY(p) ← PRIORITY(RCHILD(p));
      REMPTYNODES(p) ← REMPTYNODES(p) + 1;
      send(p_{i+1}, 'xmax-adjust(RCHILD(p)')
    end
  else
    begin
      DATA(p) ← DATA(LCHILD(p));
      PRIORITY(p) ← PRIORITY(LCHILD(p));
      LEMPTYNODES(p) ← PRIORITY(p) + 1;
      send(p_{i+1}, 'xmax-adjust(LCHILD(p)')
    end
else
  begin
    DATA(p) ← null;
    PRIORITY(p) ← -1
  end
```

Remark 3 The elements stored in the apex nodes are not ranked in any particular order. This speeds up the insertion process, but lead to O(M) time for deletion. It is possible

Proof From algorithms for *XMAX*, *zmax-adjust*, *INSERT*, and *insert-adjust*.

Lemma 4 *The element with the highest priority is always reported by operation* XMAX.

Proof: From lemma 4 and the first part of algorithm for *XMAX*.

Definition 10 *A partition induced by LEMPTYNODES and REMPTYNODES fields of apex i is the set of all nodes which are reachable by partition from apex i.*

5 Retrieval at Percentile Levels

One of the most important advantages of banyan heap machine over the binary heap machine is that it is possible to retrieve elements at different percentile levels. In this section we derive formulas for the percentile level of the element reported by operation *XMAX* for different cases.

Definition 11 *An element removed from a banyan heap is at percentile c if at least c percent of the elements stored in the heap have priority less than or equal to the priority of the deleted element.*

We define $REMPTYNODES_i$ and $LEMPTYNODES_i$ to denote respectively the value of REMPTYNODES field and LEMPTYNODES field of apex i. The proof of the following 4 lemmas are immediate from the definitions of REMPTYNODES and LEMPTYNODES.

Lemma 5 *The total number of null nodes which are reachable by partition from apex i is* $REMPTYNODES_i + LEMPTYNODES_i$.

Lemma 6 *If an MxM partitioned rectangular SW-banyan banyan heap is full up to apex d then*

$$\sum_{j=1}^{d}(REMPTYNODES_j + LEMPTYNODES_j) = 0.$$

Lemma 7 *In an MxM rectangular SW-banyan, the total number of null nodes reachable by partition from apexes 1 through d, written NULLNODES(M,d), is given by:*

$$NULLNODES(M,d) = \sum_{i=1}^{d}(REMPTYNODES_i + LEMPTYNODES_i) + K.$$

where K is the number of null apexes i, ($i \leq d$).

Lemma 8 *The total number of non-null nodes in a MxM partitioned rectangular SW-banyan, written NONNULLNODES(M,M), is* $M(\log M + 1) - NULLNODES(M,M)$.

Lemma 9 *The total number of partitions of depth k in a full partitioned banyan heap up to apex d, written NP_k, is given by:*

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor$$

where $NP_1 = \lfloor \frac{d}{2} \rfloor$.

Proof. We prove this lemma by induction on d.

Basis: For $d = 1$, $NP_j = 0$ for all $1 \leq j \leq \log N + 1$.

Induction: Assume that

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

Consider d+1. Either d is even or it is odd. If d is even then apex d+1 is the root of a partition with depth 1. Therefore,

$$NP_k = \lfloor \frac{d + 1 - \sum_{j=2}^{k-1} NP_j - (NP_1 + 1)}{2} \rfloor,$$

that is,

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

For the case that d is odd we consider two subcases: Apex $d + 1$ is either the root of a partition of depth k or it is the root of a partition with depth h, $(h < k)$. If $d + 1$ is the root of a partition of depth d then we will have

$$NP_k + 1 = \lfloor \frac{d + 1 - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

Since d is even then we have

$$NP_k + 1 = 1 + \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor,$$

or

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

If apex $d + 1$ is the root of a partition with depth h where $1 \leq h < \log N + 1$ and $h \neq k$ then we have

$$NP_k = \lfloor \frac{d + 1 - \sum_{1 \leq j \leq k-1, j \neq h} NP_j - (NP_h + 1)}{2} \rfloor,$$

or

$$NP_k = \lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \rfloor.$$

Lemma 10 *The total number of non-null nodes in a full MxM partitioned rectangular SW-banyan up to apex d, written size(M,d), is given by:*

$$size(M,d) = \sum_{k=0}^{M \log M} 2^k + \sum_{j=1}^{d} NP_j 2^j$$

Proof From lemma 9.

Lemma 11 *If an MxM rectangular SW-banyan partitioned heap is full up to apex d then the element stored at apex 1 is at percentile level*

$$\frac{(2M-1) * 100}{size(M,d)}.$$

Proof $size(M,d)$ is the total number of nodes in an MxM rectangular SW-banyan heap which is full up to apex d and $2M-1$ is the total number of nodes in partition 1.

Lemma 12 *In an MxM partitioned rectangular SW-banyan heap which is full up to apex d, if operation XMAX investigates i,(i<d), non-null apexes then the percentile of the reported element is*

$$\frac{size(M,i) * 100}{size(M,d)}.$$

Proof From lemma 10.

Lemma 13 *If operation XMAX examines apexes 1 through d in an MxM rectangular banyan heap then the percentile of the reported element is smaller than or equal to*

$$\frac{size(M,d) * 100}{(size(M,M) - NULLNODES(M,M))}.$$

Proof If a banyan heap is full up to apex d then by lemma 12 the percentile of the element reported by operation XMAX is

$$\frac{size(M,d) * 100}{size(M,d)}.$$

By lemma 7, this can be written as

$$\frac{size(M,d) * 100}{(size(M,M) - NULLNODES(M,M))}.$$

But the banyan is not full up to apex d and therefore some of the nodes which are reachable from apexes 1 to d are null. Let F be the total number of such nodes. Therefore the percentile of the element reported by XMAX operation will be

$$\frac{(size(M,d) - F) * 100}{(size(M,M) - NULLNODES(M,M))},$$

This proves the lemma.

Remark 5 Priority queue can also be implemented as banyan heap. A partitioned banyan heap can be converted into a banyan heap by an operation called *adjust*. $M \log M - 2$ *adjust* operations are broadcast by *XMAX* operation when it inserts an element into an empty partition. These *adjust* operations cause some the elements in the nodes of those partitions which are reachable from apex i to move up and fill up the nodes of partition i. As a result, all the nodes whose contents (empty or non-empty) have been moved by *adjust* operation become reachable by partition from apex i. It should be noted that some of the *adjust* operation initiated at processor p_0 by XMAX operation may not have any effect on the structure of the heap.

The advantage of banyan heap over partitioned banyan heap is that it allows a more uniform distribution of data elements among the partition in the heap and leads to a more uniform increase in the percentile level of the reported element as the number of examined apexes is increased.

Algorithms for *XMAX*, *zmax-adjust* and *insert-adjust* are the same as partitioned banyan heap. The operation *INSERT* and the new operation *adjust* is described in detail in [4].

6 Third implementation

In the first two implementations, processor p_0 becomes a bottleneck because all the operations must pass through this processor. This limits the potential concurrency of these two implementations and prevent them from fully utilizing the hpercube topology. The second mapping, described in the previous section, leads to an implementation that tends to distribute the load among the processors more evenly and achieve greater concurrency than the other two implementations.

The second mapping maps each column of the banyan heap into a unique processor, a processor whose number is the same as the number of that column. Thus, if a node is assigned to processor x then the left child of that node is stored in processor x and the right child is stored in a processor which is adjacent to processor x and whose address is m where m is the integer computed by inverting the i^{th} most significant bit in the binary representation of i. The nodes in a given column of the banyan heap is stored sequentially in an array of size $\log M + 1$ in the local memory of its processor.

The algorithmic principles of XMAX and INSERT operations are the same as the second implementation. But since now columns of the heap are stored in distinct processors in the hpercube, the operation of computing the leftmost partition that has at least one empty node (as part of INSERT operation) and the operation of locating the apex with the highest priority (as part of XMAX operation) can be performed in $O(\log M)$ rather than $O(M)$. This leads to the total response time of $O(\log M)$ for XMAX operation as will be discussed later in this section.

To implement the priority queue operations we need two operations: Global broadcast and Global min. Global broadcast is used to distribute a message to all the node in the hpercube, and global min is used to compute the minimum of a list of items which are distributed among the processors of the hpercube [6]. Below we describe algorithms for

these two operations.

Global broadcast: The procedure is based on the idea of fanout tree in which each node sends the message to all of its neighbors which have not already received it. The fanout tree rooted at node 0 for a 4 dimensional hpercube is given in figure 7. In the fanout tree each node sends the data to all neighbors on the list of neighbors of that node after the node they receive it from . Nodes in each of the list of neighbors are ordered by increasing node number. The fan out tree rooted at any other node r can be computed by taking the exclusive or of any node in the tree with r. See figure 8 for the fanout tree rooted at node 13. The following theorem due to Brandenburg and Scott [7] suggests an efficient implementation for this algorithm. The theorem is followed by the code executed by node x upon receiving a message from node z.

Theorem 4 *If the root r sends to everybody and each other node z, which receives a message from z, sends it on to all neighbors* $y = x + 2^j$ *such that* $2^j > x + z$, *then the resulting fanout tree is the same as that obtained from exclusive or with r of tree rooted at zero.*

```
p ← 2^d
bit ← 2 * x + z;
for i ← 1 to d do
begin
    if (bit ≠ p ) then
    begin
        node ← x + bit;
        send the message to node NODE;
    end
    bit ← 2 * bit
end
```

Global Min: The algorithm for this operation is also based on the idea of fanout tree. In the fanout tree, each node computes the minimum of its current item and those of its children and then pass it to its parent. The execution of the code for Global Min at different processors will be started when a message is received from the root of the corresponding fanout tree requesting the computation of the minimum. Below is the code for Global Min executed by the nodes of the fanout tree upon receiving the Global Min signal.

```
p ← 2^N
bit ← p/2
while( bit > NODE) do
begin
        receive a value in VAL from one of the
        children of node NODE;
        x ← min(VAL,x);
        bit ← bit/2
end
nnode ← (root + NODE)
PARENT ← (nnode + bit)
send x to PARENT
```

Operations XMAX and INSERT for this implementation are described below.

INSERT: An INSERT operation initiated at node x first computes the address of the leftmost apex y whose partition has at least one empty node (using Global Min operation). The new pair of element-priority pair is then sent from node x to the node which contains apex y. The pair will be inserted into the partition whose root is apex y according to the procedure described for the second implementation.

The problem with the INSERT operation as given above is that an insertion operation may find a partition (reported to have empty positions) full when it tries to insert a pair into that partition, and therefore blocked and unable to proceed. This is caused by allowing several INSERT operations to search the list of apexes concurrently for their point of insertions, which as a result more than one INSERT operation may receive the same apex whose corresponding partition has only one empty position as the point of insertion. In this situation a new INSERT operation may be reissued at the node which blocked the insertion.

XMAX: An XMAX operation initiated at node x first determines the apex which contains the element with the highest priority (Using Global Min operation) and then send an operation, called *adjust*, to the node containing that apex. Operation *adjust* first report the element to node x, and then adjust the banyan heap as described in the previous section.

The problem with *XMAX* operation as described above is that due to concurrent access to the list of apexes by several processors, the same apex may be reported to several XMAX operations as the holder of the element with the highest priority. This may lead to the situation in which elements returned and subsequently deleted by some of the XMAX operation do not have the highest priority in the banyan heap at the time of removal. One solution to this problem is to send all the XMAX operations (issued at different nodes) to node p_0 and let node p_0 execute them sequentially. This limits the amount of concurrency

obtainable in the system. The amount of obtainable concurrency starts to decrease due to this strategy when the number of elements in the banyan heap exceeds $2^{\log M+2} - 1$ for the first time.

Theorem 5 *Operation XMAX requires $O(\log M)$ time to complete.*

Proof: Operation XMAX consist of 3 parts:

1. finding the apex containing the element with the highest element.
2. reporting the element to the node initiated the operation.
3. adjusting the banyan heap

Each of these steps requires $O(\log M)$ time and hence the total time of $O(\log M)$ for XMAX operation.

Theorem 6 *Operation INSERT requires $O(\log M)$ time to complete.*

Proof: Similar to theorem .

Theorem 7 *An XMAX operation needs no more than $2(M-1)+2*\log M$ communications.*

Theorem 8 *Third implementation offers $O(M/\log M)$ throughput.*

Proof: The third implementation can perform M operation at a time and each operation requires $O(\log M)$ time, and hence $O(M/\log M)$ throughput.

In what follows we first define few terms and then introduce the concept of consistency for concurrent data structure.

Definition 12 *Timestamp of an operation is the time at which the operation is initiated at a node and timestamp of a communication is the timestamp of the operation which generated that communication.*

Definition 13 *Operations $O_1, O_2, O_3, ...,$ and O_q with timestamps $t_1, t_2, t_3, ...,$ and t_q are said to be a sequence of operations if $t_1 < t_2 < .... < t_q$ and operations O_i, $1 < i \leq q$ are the only operations issued between t_1 and t_q .*

Definition 14 *A concurrent data structure is said to be consistent if the concurrent execution of any sequence of operations on the data structure gives the same result as executing the operations sequentially.*

Theorem 9 *The third implementation is not consistent*

Proof: Consider a sequence of XMAX operations waiting in node p_0 for execution. Due to the fact that these XMAX operations are executed sequentially in increasing order of their timestamps, it is quiet possible for an INSERT operation which have a larger timestamp than any of the XMAX operations to be executed before all the XMAX operations are completed.

Theorem 10 *The third implementation is consistent if an access made to an apex by an operation is not started unless all the operations with the lower timestamps have completed all their M+1 accesses to the apexes.*

Below we describe a procedure for making the third implementation consistent.

An operation is first recorded by all the processors in the hpercube. The processor which initiated the operation broadcast that operation together with its timestamp to all the processors using its fan out tree. The operation and its timestamp is added to the list of incomplete operations maintained by resident executive of each processor. After the operation is added to the list of incomplete operations by a processor, that processor send a notification signal to the processor initiated the operation. The notification signals issued by the processors are combined according to the fan out tree rooted at the processor which initiated the operation; no processor sends a notification signal to its parent in the fan out tree unless all its children have noted that operation. After the root processor received all the notification signals, the execution of the operation starts. During the execution of the operation no access is made to an apex unless all the operations with lower timestamps have completed their access to that apex. When an operation completes its last access to the apexes, it asks all the executives to remove that operation from their list of incomplete operations. The removal operation is performed in the same fashion as the operation of recording an operation. After an operation is removed from the list of incomplete operations of an executive, the next pending operation will be allowed to perform its access to the corresponding apex.

7 Conclusion

We have proposed three concurrent data structures for implementing priority queues on a distributed-memory message passing multiprocessor with hpercube topology. These concurrent data structures can be used by any application running on the hpercube without Worring about all the necessary communications and synchronizations. Priority queue operations each requires $O(\log M)$ time to complete, but since M operations may be initiated simultaneously at different processors, $O(M/\log M)$ throughput is achievable as compared to $O(1)$ throughput for sequential data structure.

8 References

1. J. Biswas and J. C., "Simultaneous Update of Priority Structures," Proceedings of International Conference on Parallel Processing, August 1987, pp. 124-131.

2. M. J. Carey and C. D. Thompson, "An efficient Implementation of Search trees on $\lceil \log N + 1 \rceil$ processors," IEEE Transactions on Computers, Vol. C-33, No. 11, Nov. 1984, pp. 1038-1041.

3. Y. Saad and M. Schultz,"Topological Properties of Hpercubes," IEEE Transactions of Computers, Vol. 37,No. 7, July 1988, pp. 867-872.

4. M. R. Meybodi, " New Designs for Priority Queue Machine," Computer Science Technical Report, Ohio University, Athens, Ohio, 47501.

5. M. R. Meybodi, "Tree Structured Dictionary Machine for VLSI," Computer Science Technical Report, Ohio University, Athens, Ohio, 47501.

6. C. Moler and D. S. Scott, "Communication Utilities For The iPSC," iPSC Technical Report, August 1986.

7. J. E. Brandenburg and D. S. Scott, "Embeddings of Communication Tree and Grid into Hpercubes," iPSC Technical Report, August 1986.

8. V. N. Rao and V. Kumar, "Concurrent Access of Priority Queue," IEEE Transactions on Computers, vol 37, No. 12, Dec. 1988, pp. 1657-1665.

9. W. J. Dally, A VLSI Architecture for Concurrent Data Structures, Kluwer Academic Publishers, 1987.

10. M. J. Quinn, Designing Efficient Algorithms for parallel Computers, McGraw Hill, 1987.

11. C. D. Thompson, "The VLSI Complexity of Sorting," IEEE Transactions on Computers, Vol. C-32, No. 12, Dec. 1983, pp. 373-386.

12. A. R. Omondi and J. D. Brock, "Implementing a Dictionary on Hypercube Machine," Proceedings of International Conference on Parallel Processing, August 1987, pp. 707-709.

13. K. H. Cheng, "Efficient Designs of Priority Queue," Proceedings of International Conference on Parallel Processing, August 1988, pp. 363-366.

14. J. H. Chang, O. H. Ibarra, M. J. Chang, and K. K. Rao, " Systolic Tree Implementation of Data Structures," IEEE Transactions on Computers, Vol. 37, No. 6, June 1988, pp. 727-735.

15. J. D. Ullman, Computational Aspect of VLSI, Computer Science Press, 1984.

16. T. A. Standish, Data Structures Techniques, Addison Wesley, 1980.

17. E. Horowitz and A. Sahni, Fundamentals of Data structures, Computer Science Press, 1983.

18. H. T. Kung and P. L. Lehman, "Concurrent Manipulation of Binary Search Trees," ACM Transactions on Database Systems, Vol. 5, No. 3, Sept. 1980, pp. 354-382.

19. P. L. Lehman and S. B. Yao, "Efficient Locking for Concurrent operations on B-Trees," ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, pp. 650-670.

20. C. S. Ellis, "Concurrent Search and Insertion in AVL Trees," IEEE Transactions on Computers, Vol. C-29, No. 9 September 1980, pp. 811-817.

21. C. S. Ellis, "Concurrent Search and Insertion in 2-3 Trees," Acta Information, Vol. 14, 1980, pp. 63-86.

22. J. L. Bentley and H. T. Kung, "A Tree Machine for Searching Problems," Proceeding of the International Conference on Parallel Processing, 1979.

23. C. E. Leiserson, "Systolic Priority Queues," Dept. of Computer Science, Carnegie Melon University, Pittsburgh, PA, Report CMU-CS-115,1979.

24. T. A. Ottmann, A. L. Rosenberg, and L. J. Stockmeyer, "A Dictionary Machine for VLSI," IEEE Transaction on Computers, vol. c-31, No. 9, Sept. 1982, pp. 892-897.

25. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI), " Proceedings of Symposium on Sparse Matrix Computations and their Applications, November 1978, pp. 256-282.

26. A. K. Somani and V. K. Agarwal, "An Efficient VLSI Dictionary Machine, " Proceedings of 11th Annual International Symposium on Computer Architecture, 1985, pp. 142-150.

27. M. J. Atallah and S. R. Kosaraju, "A Generalized Dictionary Machine for VLSI," IEEE transactions on Computers, Vol. C-34, No. 2, Feb. 1985, pp. 151-155.

28. H. Schmeck and H. Schroder, "Dictionary Machines for Different Models of VLSI," IEEE Transaction on computers, Vol. C- 34, No. 5, May 1985, pp. 472-475.

29. A. L. Fisher, "Dictionary Machines with Small Number of Processors," Proceedings of International Symposium on Computer Architectures, 1984, pp. 151-156.

30. D. Knuth, The Art of Computer Programming, Vol. 3, 1973.

31. E. M. Reingold and W. J. Hansen, Data Structures, Little, Brown and Company, 1983.

32. N. J. Nilsson, Problem Solving Methods in Artificial Intelligence, McGraw Hill, 1971.

33. F. Dehne and N. Santoro, "Optimal VLSI Dictionary Machines on Meshes," Proceedings of International Conference on Parallel Processing, August 1987, pp. 832-840.

34. L. R. Goke and G. L. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," Proceedings of the First Annual Symposium on Computer Architecture, 1973, pp. 21-28.

35. A. L. Tharp, File Organization and Processing, John Wiley and Sons, 1988.

36. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.

37. R. Nix, "An Evaluation of Pagodas," Tech. Rep. 164, Computer Science Dept. Yale Univ.

38. D. D. Sleator and R. E. Tarjan , "Self Adjusting Heaps," SIAM J. Comput. Vol. 15, No 1, Feb. 1986, pp. 52-59.

39. J. Vuillemin, "A Data Structure for Manipulating Priority Queues," Comm. ACM, Vol. 21, No. 4, April 4, pp. 309-315.

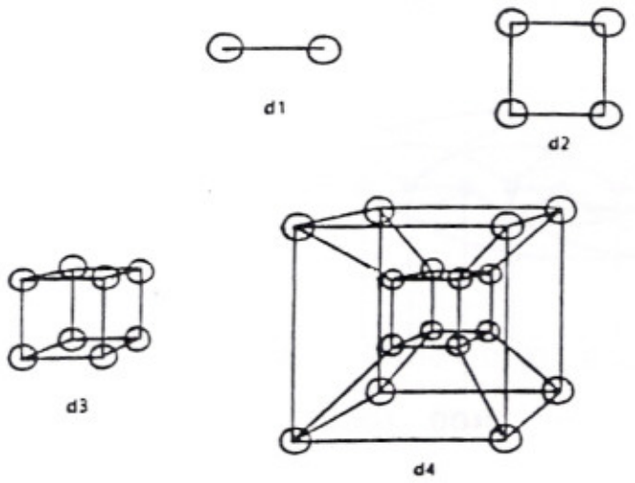

Figure 1.

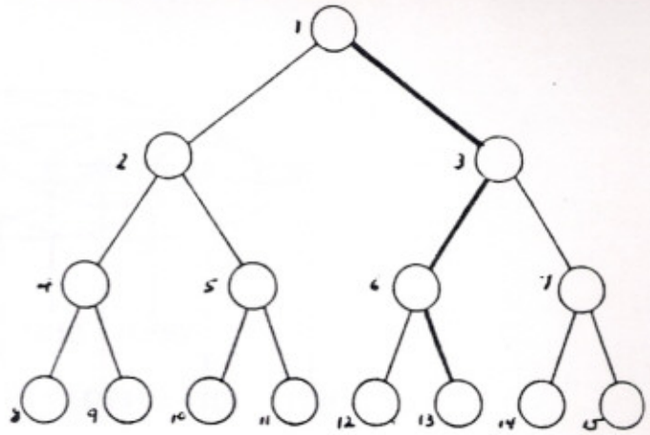

Figure 2.

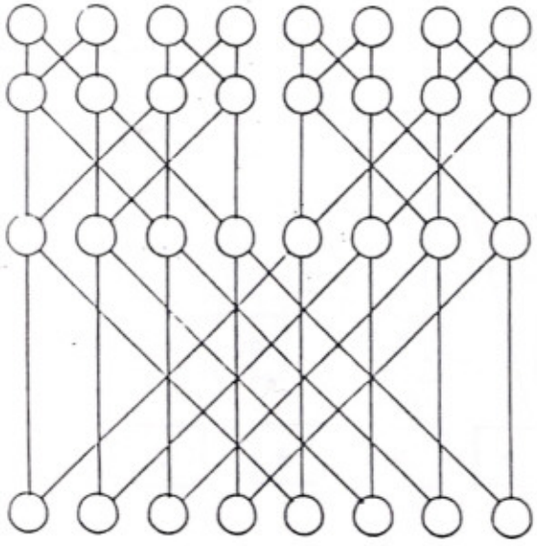

Figure 3.

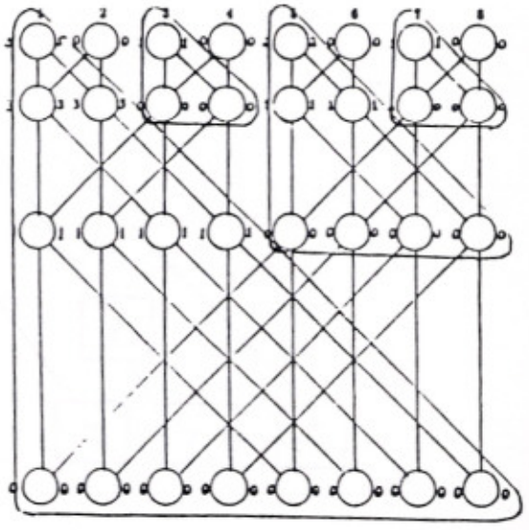

Figure 4.

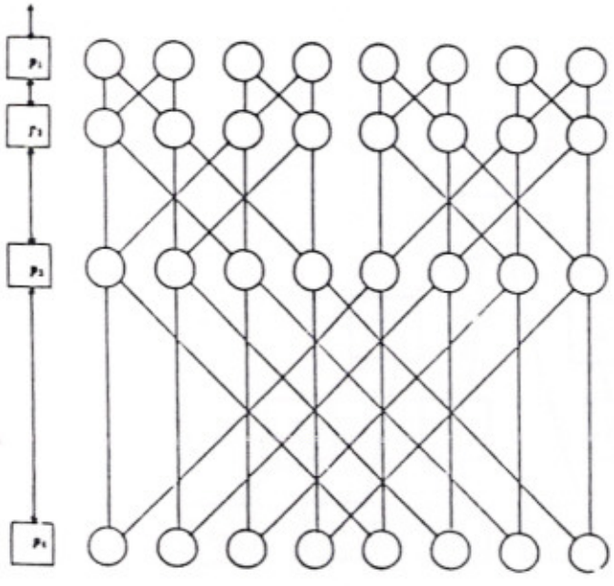

Figure 5.

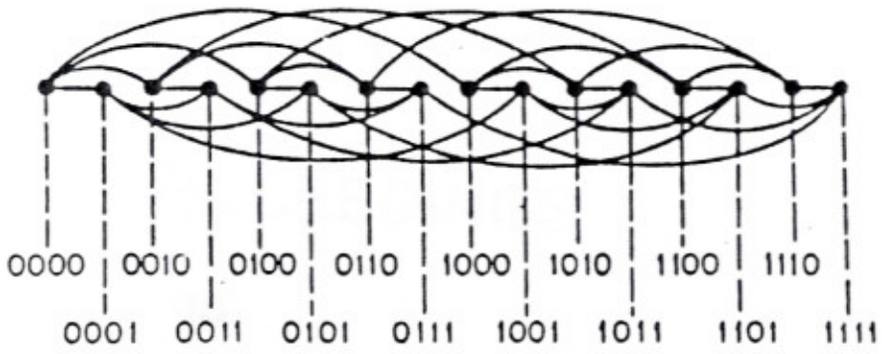

Figure 6.

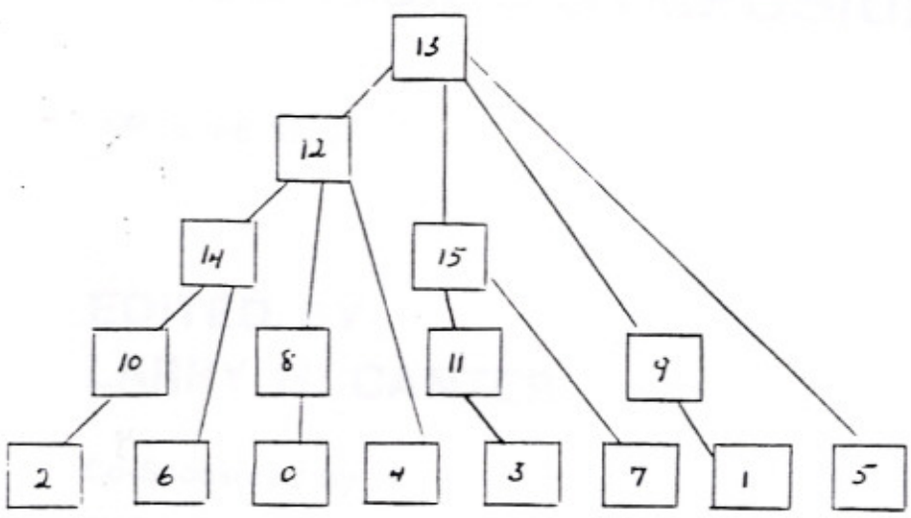

Figure 7.

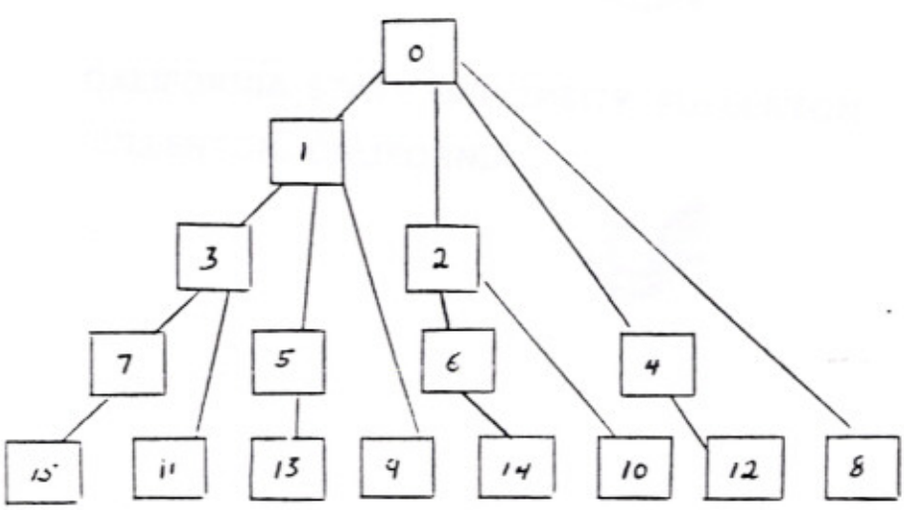

Figure 8.

FOURTH ANNUAL PARALLEL PROCESSING SYSPOSIUM

Sponsored by the Orange County IEEE Computer Society,
and California State University, Fullerton

Location: California State University Fullerton
at the University Center

Dates: April 4, 5, 6 1990, Time 8:30 - 5:30 Each Day

Symposium General Chairman:	Dr. Larry H. Canter Computer Systems Approach, Inc.
Technical Co-Chairpersons:	Dr Nader Bagherzadeh, UCI Sandeep Gulati, JPL
Program Committee Chairman:	Dr. V. K. Prasanna Kumar, USC Dr. John Clymer, CSUF Dr. Mary Eshaghian, USC Dr. Dough Blough, UCI Kichol Kim, USC Suresh Chalasani, USC Wei Ming Ling, USC Dr. Alireza Kavianpour, UCI Dr. Sasamma Barua, CSFU
Conference Consultant:	Dr. Howard Jelinek, EDA
Conference Advisor:	Alex Williman, RI
Commercial Sessions Chairman:	Bill Pitts, Toshiba America
Campus Coordinator:	Dr. Michael Singh, CSULB
Student Presentations Chairperson:	Dr. Sammana Barua, CSUF
Staff Coordinator:	Corrinne Yu, Cal Poly Pomona
Board of Directors:	George Westrom, Odetics Dr. Arnold Miller, Pres Xerox Electronics Division (Retired)

Company Sponsors:

Hughes Aircraft Company	Rockwell International
Intel Corporation	Motorola Corporation
Levco Sales	Alliant Corp