PARLE ’92
Parallel Architectures and Languages Europe

4th International PARLE Conference
Paris, France, June 15-18, 1992
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest
Concurrent Data Structures for Hypercube Machine

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

Abstract
To efficiently implement parallel algorithms on parallel computers, concurrent data structures (data structures which are simultaneously updatable) are needed. In this paper, three implementations of a priority queue on a distributed-memory message passing multiprocessor with a hypercube topology are presented. In the first implementation, a linear chain of processors is mapped onto the hypercube, and then a heap data structure is mapped onto the chain, where each processor stores one level in the heap. A similar approach is taken for the second implementation, but in this case, a banyan heap data structure is mapped onto the linear chain of processors. Again, each processor in the chain becomes responsible for one level of the data structure. For the third implementation, the banyan heap data structure is again used, but the mapping is not onto linear chain of processors. Instead, the banyan heap is mapped onto processors column by column, so that the algorithm can make better use of the concurrent processing capabilities of the hypercube topology in order to reduce bottlenecking in the first processor, an effect noted in the use of the linear chain employed by the first two implementations. 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, Hypercube, Banyan Heap, Parallel Algorithm

1 Introduction
Priority queue data structure is a very important data structure which has found application in various situations such as: discrete event simulation systems, time-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)):
\[ P \leftarrow P \cup \{(x, p(x))\}. \]
Response is null.

XMAX:
\[ P \leftarrow P \setminus \{(p_{\text{max}})\} \text{ where } (p_{\text{max}}) \text{ is the pair with the highest priority}. \]
Response is \( P(p_{\text{max}}) \).
message passing multiple
on heap data structure.
embedded into hypercube
structure. Then differ in
advantage of banyan and
router elements at different
places.

An d dimensional hypercube
another to form a d din
and only if the binary processor node in the k processor node in

The rest of this paper of
priority queue on a di
and banyan heaps. The
banyan heap data struc
ted a retrieved element.
In this paper is presented.

2 First Implemen
tation
This implementation
linear chain of a proce
way as to preserve the
chain are mapped into
known tecmique to ob
such as
\( b_{i+1} = b_i + b_{i+1} \)
if \( i < d \)
in the chain is mapped in
one level of the heap
not be discussed here.
Processor \( P_i \), \( 0 \leq i \leq 3 \)
has 6 fields: DATA, FB
ODES. For a node, the
priority associated with
the leftchild and right
he number of null nods
node. Initially, the DA
all the nodes are set to
nodes at level i are ini
are inserted into or del
is used by \texttt{INSERT} op
the insertion process. L
processor. This happens
are non-empty.

The process running:
The first part belongs to
The second part, called
ations. An executive cas
operations are initiated
message passing multiprocessor with hypercube 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 hypercube computer is briefly described below.

An $d$ dimensional hypercube machine consists of $2^d$ processor nodes interconnected to one another to form a $d$ 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 hypercube has a processor and a local memory for that processor. The processors work independently and asynchronously and communicate by passing messages.

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 system. Section 3 defines banyan graphs and banyan heaps. In section 4 we discuss an implementation of priority queue based on banyan heap data structure. Section 5 derives analytical formula for the percentile level of a retrieved element. In section 6, the second banyan heap based implementation of priority queue is presented. 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 hypercube. The use of Gray codes is one known technique to obtain a mapping which preserves the proximity property. This technique can be explained as follows: If the binary representation of the node number in the chain is $b_d ... b_2 b_1 b_0$, then it is mapped into the node with number $c_d ... c_2 c_1 c_0$, where $c_i = b_i + b_{i+1}$ if $i < d - 1$, and $c_i = b_i$, if $i = d - 1$. For example, node number 26 ($0110101$) in the chain is mapped into node $23(0101111)$ in the hypercube. 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 $n_i$ ($0 \leq i \leq n - 1$), of the chain stores the $i^{th}$ level of the heap. Each node has 8 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 hold 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 the 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.

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 hypercube. The second part, called executive, is responsible for the execution of the priority queue operations. 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 hypercube. An operation issued by a process is communicated to the executive of that process. The executive then sends 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 pipelined 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 instructions either from another executive or from the application part of the process to which it belongs. Now we describe operations \(XMAX\) and \(INSERT\).

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 traverses the heap from the top to perform the restructuring process and, unlike the conventional algorithm, it does not necessarily expand the heap level by level.

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

\[ XMAX \text{ operation: When the executive of } p_0 \text{ receives operation } XMAX \text{ from another executive or from the application part of its own process it generates an operation called adjust which propagates through the chain of processors and converts the binary tree (after the root is removed) into a heap. With respect to operation } XMAX, \text{ all the processors except processor } p_0 \text{ perform the same set of actions. We describe operation } XMAX \text{ as follows:} \]

\[ \text{processor } p_0: \text{ When processor } p_0 \text{ receives operation } XMAX, \text{ it reports the element with the highest priority to the processor that issued the } XMAX \text{ operation and then sends operation adjust to processor } p_1. \]

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

The response time for \(XMAX\) is \(O(\log N)\), because it takes \(O(\log N)\) time for \(XMAX\) operation to reach processor \(p_0\) by both the \(XMAX\) processors. This is due to the place within the tree.

Remark 1 Infor nodes are used by the insert \(INSERT\) operation last, the index of the deepest level first and \(p = \log_2 n\) representation of heap. At level \(i \geq 1\) or left (if \(0\)), \(i\) is a \(k\)-ary heap which we use \((k + 1)\) \(b\) heap. The advantage memory space sequential algorithm left and right sub sections.

3 Banyan

A banyan graph span from any base it and an apex is an apex nor a base.

An \(L\)-level banyan of length \(L\). The \(L\) of edges. By dom.

Banyan graph, the node. If there is a is the parent of \(y\),

Definition 1 A \(\text{A and have identical\space vectors, } F = (F_{ij}), \text{ respectively, where}\)

In a uniform banyan graph, \(F = (F_{ij})\), for same \(i\).

Definition 2 If a rectangular. If \(x \leq L - 1\), for some \(c\)
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 \texttt{XMAX} and \texttt{INSERT} operations is \( O(1) \), independent of the length of the chain of processors. The execution time for each of the \texttt{XMAX} or \texttt{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 \texttt{XMAX} operation.

Remark 1 Information stored in the \texttt{EMPTYNODES} and \texttt{REMPYTHNODES} fields of the nodes are used by \texttt{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 \texttt{INSERT} operation [5,8]. This requires the system to maintain two pieces of information: \texttt{last}, the index of the last nonempty node of the heap and \texttt{first}, the index of the leftmost node in the deepest level of the heap which contains at least one nonempty node. Let \( I = \texttt{last} - \texttt{first} \) and \( p = \log \texttt{last} \). The number \( I \) can be expressed as a \( p \)-bit binary number. The binary representation 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 from the left of \( I \) to decide whether to go to the right (if \( I \)) or left (if \( 0 \)). Root of the heap is at level 1. This procedure can be extended to apply to a \( k \)-ary heap where \( k = 2^k \). If \( I = \texttt{last} - \texttt{first} \) and \( p = \log \texttt{last} \), then at level \( d \) of the heap we use \( (1 + 1)^{th} \) 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 used in this report 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 becomes 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 \( L + 1 \) levels of nodes and \( L \) levels 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 \( y \) 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 \( \leq i \) \( \leq L - 1 \)), then the banyan is called rectangular. If \( s_{i+1} \neq f_i \) 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 ≤ i ≤ L),
and f_i = d, (1 ≤ i ≤ L - 1), for some constant d.

Having introduced the necessary notations 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 1 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 dimensional cube
where 2^n = log M + 1. M is the number of apexes. The restriction to an 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 fields of all the nodes are set to null and the priority fields of all the
nodes are set to -1. To initialize these fields, first partition the heap into M disjoint binary
trees and then use the same rule as for the first design to initialize the LEMPTYNODES and
REMTYNODES 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 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 2.

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 i at level j + 1 is reachable by partition
from node k at level j if node i either has non-zero LEMPTYNODES and it is the right child
of node k, or has non-zero LEMPTYNODES and it is the left child of node k. 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 reachability of the nodes will change
by a later deletion operation initiated at some other apexes. Reachability does not imply
reachability by partition.

4 Second Implo
The effective implement

4 Second Implementation

The effective implementation of banyan heap on hypercube requires efficient mapping of the banyan structure among the processors of the hypercube. We examine two such mappings. The first mapping is obtained by first mapping the banyan into a linear chain and then embedding the chain into the hypercube (See figure 3). The second mapping is obtained by collapsing columns of the banyan into single processors [34]. That is 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 hypercube, those having labels that differ exactly in the \( i \text{th} \) digit. See figure 8 for an example. In this section we consider the first mapping.

Before we give a detailed description of the operations \textit{XMAX} and \textit{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 \( n_{ij} \) denote the \( i \text{th} \) node on the \( j \text{th} \) level of the banyan where \( 0 \leq i \leq M, 0 \leq (\log M + 1) \). Then \( n_{ij} \) is connected to two nodes \( n_{(i+1)j} \) and \( n_{(i+1)m} \), where \( m \) is the integer found by inverting the \( i \text{th} \) most significant bit in the binary representation of \( i \). We call nodes \( n_{(i+1)j} \) and \( n_{(i+1)m} \) the right and left children of node \( n_{ij} \), respectively. Therefore, the left child and right child of node \( i \) at level \( k \) are stored respectively in the \( i \text{th} \) location and \( m \text{th} \) location of the array residing in the local memory of processor \( k + 1 \).

All the \textit{XMAX} and \textit{INSERT} operations initiated at different processors are sent to the head of the embedded chain and are executed in a pipelined manner. When operation \textit{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 \textsc{REMTYNODES} and \textsc{LEMTYNODES} fields of the apexes in \( O(M) \) time. It then pushes the element requested to be inserted down the banyan heap using operation \textit{insert-adjust}. The operation \textit{insert-adjust} pushes the element down (along the paths from the root of the partition to the base) until it finds its correct position. This requires \( O(\log M) \) time. Thus \textit{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 \textit{Send}(<processor>,<instruction>) sends instruction \<instruction\> to processor \<processor\> for execution. The execution of \textit{receive}(<processor>,<instruction>) 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 \textit{INSERT}(p, (item,priority)) by processor \( p_0 \), it executes the following codes. The letter \( p \) is the address of the least cost apex, and (item,priority) is the pair requested to be inserted.

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

Processor pi, (2 ≤ i ≤ L), upon receiving insert-adjust(p_i, (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

            end;

XMAX operation first locates the element to be inserted in the heap. When XMAX(p, p) is the address of the left child of the element and p the address of the right child.
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
end
send(p_{p+1}, 'insert-adjust(p', (DATA(p), PRIORITY(p)))
DATA(p) ← item; PRIORITY(p) ← priority

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
      end
    else
      begin
        DATA(p) ← item; PRIORITY(p) ← priority
      end
  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_i, it executes the following codes, where 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(p2, ((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(p2, 'xmax-adjust(RCHILD(p'))
    end
  else
    begin
      DATA(p') ← DATA(LCHILD(p'));
      PRIORITY(p') ← PRIORITY(LCHILD(p'));
      REMPTYNODES(p') ← REMPTYNODES(p') + 1;
      send(p2, 'xmax-adjust(LCHILD(p'))
    end
  end
else
  begin
    DATA(p') ← null;
    PRIORITY(p') ← -1
  end

Processor p1, (1 ≤ i ≤ 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));
      REMPTYNODES(p) ← PRIORITY(p) + 1;
      send(p_{i+1}, 'xmax-adjust(LCHILD(p))'
    end

Remark 3 The elements
This speeds up the insertion
insert the elements in such
at the leftmost apex in w
O(M) time. This method
spent to find the correct i
with zero REMPTYNODES
we have used the first app

From the properties of the following results.

Theorem 1 Operation X

Theorem 2 Operation II

Theorem 3 The second

Lemma 1 a) The insert-
zero REMPTYNODES as
finds a null node to insert

Proof a) If operation zme
REMPYNODES fields c
both REMPTYNODES as
the algorithm for INSERT
of REMPTYNODES and

Remark 4 Deletion of an
partitions whose nodes ar
operation causes the zme
of the leaf nodes of partit
operation at the previous
REMPYNODES or LEN
from apex i and will be fi
nodes that may become r
(L + 1) − D, where D

Lemma 2 Zero REMPT
that all the nodes in the e

Proof From remark 4.
Remark 3 The elements stored in the apex nodes are not ranked in any particular order. This speeds up the insertion process, but leads to $O(M)$ time for deletion. It is possible 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 REMP TYNODES or zero LEMP TYNODES. In the algorithms presented above we have used the first approach. The second approach will be reported in another paper.

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 LEMP TYNODES and zero REMP TYNODES. b) The insert-adjust operation always finds a null node to insert its element.

Proof a) If operation $zmax-adjust$ encountered a non-null node with LEMP TYNODES and REMP TYNODES fields equal to zero then it must have been initiated from an apex with both REMP TYNODES and LEMP TYNODES 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 LEMP TYNODES and REMP TYNODES.

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 $zmax-adjust$, 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 $zmax-adjust$ operation at the previous step. The emptiness of this node now will be reflected in the REMP TYNODES or LEMP TYNODES 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 $(L+1) - D$, where $D$ is the depth of partition $i$.

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

Proof From remark 4.
Lemma 3 Apex i, \(1 \leq i \leq M\), always contains the element which has the highest priority among the elements stored in the nodes of partition i.

Proof From algorithms for \textsc{xmax}, \textsc{xmax-adjust}, \textsc{insert}, and \textsc{insert-adjust}.

Lemma 4 The element with the highest priority is always reported by operation \textsc{xmax}.

Proof: From lemma 4 and the first part of algorithm for \textsc{xmax}.

Definition 10 A partition induced by \textsc{lemptynodes} and \textsc{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 over the binary heap 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 \textsc{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 \textsc{reemptynodes}, and \textsc{lemptynodes}, to denote respectively the value of \textsc{reemptynodes} field and \textsc{lemptynodes} field of apex i. The proof of the following 4 lemmas are immediate from the definitions of \textsc{reemptynodes} and \textsc{lemptynodes}.

Lemma 5 The total number of null nodes which are reachable by partition from apex i is \textsc{reemptynodes} + \textsc{lemptynodes}.

Lemma 6 If an \textsc{mxm} partitioned rectangular SW-banyan banyan heap is full up to apex d then

\[
\sum_{j=1}^{d} (\textsc{reemptynodes} + \textsc{lemptynodes}) = 0.
\]

Lemma 7 In an \textsc{mxm} rectangular SW-banyan, the total number of null nodes reachable by partition from apexes 1 through d, written \textsc{nullnodes}(M,d), is given by:

\[
\textsc{nullnodes}(M,d) = \sum_{i=1}^{d} (\textsc{reemptynodes} + \textsc{lemptynodes}) + K.
\]

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

Lemma 8 The total number of non-null nodes in an \textsc{mxm} partitioned rectangular SW-banyan, written \textsc{nonnullnodes}(M,M), is \(M(\log M + 1) - \textsc{nullnodes}(M,M)\).

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:

\[
NP_k = \left\lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \right\rfloor
\]

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

Proof. We prove this lemma

Basis: For \(d = 1\), \(NP_1 = 0\).

Induction: Assume that

Consider \(d+1\). Either d is partition with depth 1. Then\( N\)

that is,

For the case that d is odd partition of depth k or it is of a partition of depth d then

Since d is even then we have

or

If apex \(d+1\) is the root of:

we have

or

Lemma 10 The total num

banyan up to apex d, written

Proof From lemma 9.

Lemma 11 If an \textsc{mxm} re

the element stored at apex
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

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

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,

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

that is,

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

For the case that $d$ is odd we consider two subcases: Apex $d+1$ is either the root of a partition of depth $h$ 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

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

Since $d$ is even then we have

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

or

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

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

$$NP_k = \left\lfloor \frac{d + 1 - \sum_{1 \leq i \leq h \leq d, h \neq k} NP_j}{2} \right\rfloor - (NP_{k+1} + 1),$$

or

$$NP_k = \left\lfloor \frac{d - \sum_{1 \leq i \leq h \leq d, h \neq k} NP_j}{2} \right\rfloor.$$

Lemma 10 The total number of non-null nodes in a full MaM 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 MaM rectangular SW-banyan partitioned heap is full up to apex $d$ Then the element stored at apex 1 is at percentile level

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

\[
\frac{\text{size}(M, i) \times 100}{\text{size}(M, d)}
\]

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

\[
\frac{\text{size}(M, d) \times 100}{(\text{size}(M, M) - \text{NULLNODES}(M, M))}
\]

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

\[
\frac{\text{size}(M, d) \times 100}{\text{size}(M, d)}
\]

By lemma 7, this can be written as

\[
\frac{\text{size}(M, d) \times 100}{(\text{size}(M, M) - \text{NULLNODES}(M, M))}
\]

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

\[
\frac{(\text{size}(M, d) - F) \times 100}{(\text{size}(M, M) - \text{NULLNODES}(M, M))}
\]

This proves the lemma.

Remark 5 A 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 the XMAX operation when it inserts an element into an empty partition. These adjust operations cause some of 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 the adjust operation become reachable by partition from apex i. It should be noted that some of the adjust operations initiated at processor p0 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 partitions in the heap and leads to a more uniform increase in the percentile level of the reported element as the number of examined apexes increases.

Algorithms for XMAX, zmax-adjust and insert-adjust are the same for the banyan heap. The operation INSERT and the new operation adjust are described 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 prevents them from fully utilizing the hypercube topology. For the third implementation, the banyan heap data structure is again used, but the mapping is not onto a linear chain of processors. Instead, the banyan heap is mapped onto processors column by column. See Figure 4. This leads to an implementation that tends to distribute the load among the processors more evenly and achieve greater concurrency. This mapping allows the algorithm to make better use of concurrent processing capabilities of the hypercube topology in order to reduce bottlenecks in the first processor, an effect noted in the use of the linear chain employed in the first two implementations.

In this implementation, each column of the banyan heap is assigned to 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 are stored sequentially in an array of size \( \log M + 1 \) in the local memory of its processor.

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

To implement the priority queue operations we need two operations: \( \text{Global Broadcast} \) and \( \text{Global Min} \). \( \text{Global Broadcast} \) is used to distribute a message to all the nodes in the hypercube, and \( \text{Global Min} \) is used to compute the minimum of a list of items which are distributed among the processors of the hypercube [6]. Below we describe algorithms for these two operations.

\( \text{Global Broadcast} \): The procedure is based on the idea of a 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 hypercube is given in Figure 5. 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 fanout 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 6 for the fanout tree rooted at node 13. The following theorem from Brandenburg and Scotti [7] suggests an efficient implementation for this algorithm.

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

\( \text{Global Min} \): The algorithm for this operation is also based on the idea of a fanout tree. In the fanout tree, each node computes the minimum of its current item and those of its children and then passes it to its parent. The execution of the code for \( \text{Global Min} \) at different processors is started when a message is received from the root of the corresponding fanout